

Université Fédérale



Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le 01/07/2022 (prévisionnel) par :

DANIEL LOCHE

PRÉVENTION DES FAUTES TEMPORELLES SUR ARCHITECTURES MULTICŒUR POUR LES SYSTÈMES À CRITICITÉ MIXTE

JURY

LILIANA CUCU-GROSJEAN	Directrice de Recherche, INRIA	Rapporteure
EMMANUEL GROLLEAU	Professeur des Universités, ISAE-ENSMA	Rapporteur
CLAIREE PAGETTI	Ingénierie de Recherche, ONERA	Examinateuse
SÉBASTIEN FAUCOU	Maître de Conférences, Univ. de Nantes	Examinateur
JEAN-CHARLES FABRE	Professeur des Universités, Toulouse INP	Directeur de Thèse
MICHAEL LAUER	Maître de Conférences, Univ. Toulouse 3	Co-directeur de Thèse
FRANÇOIS GEUSSE	Ingénieur, Renault SWLabs	Invité

École doctorale et spécialité :

EDSYS : Systèmes embarqués 4200046

Unité de Recherche :

Laboratoire d'analyse et d'architecture des systèmes – LAAS-CNRS

Directeur(s) de Thèse :

Jean-Charles FABRE et Michael LAUER

Rapporteurs :

Liliana CUCU-GROSJEAN et Emmanuel GROLLEAU

Souvent une évolution est une révolution sans en avoir l'R.

Pierre-Henri Cami

Table des matières

Liste des Abréviations	v
Introduction	1
1 Enjeux des Systèmes Embarqués Complexes	5
1.1 Évolutions des Systèmes embarqués	6
1.1.1 Nouveaux systèmes intelligents et connectés	6
1.2 Risques et Problématique	11
1.2.1 Sûreté de Fonctionnement Informatique	12
1.2.2 Systèmes temps-réel et Ressources partagées	14
1.2.3 Problématique et Objectifs	17
1.3 Contraintes et Hypothèses	18
1.3.1 Contexte industriel Automobile	18
1.3.2 Standards industriels et Concept de Criticité	19
1.3.3 Contraintes d'intégration	20
1.4 Grandes approches du domaine	22
1.4.1 Mécanismes de contrôle	22
1.4.2 Mécanismes Réactifs	23
1.5 Contribution de la thèse et objectifs	24
2 État de l'Art	27
2.1 Vue d'ensemble des systèmes à criticité mixte	28
2.1.1 Entre garanties et optimisation	28
2.1.2 Solutions Matérielles et Logicielles	30
2.1.3 Contrôle Statique et Dynamique	31
2.2 Mécanismes de contrôle réactif	34
2.2.1 Contrôle réactif spatial	34
2.2.2 Contrôle réactif temporel	35
2.3 Positionnement des travaux	37
3 Principe et architecture	39
3.1 Modèle basé sur des chaînes de tâches	40
3.1.1 Notion de Criticité	41
3.1.2 Modèle de Tâche et Chaînes de tâches	43
3.2 Mécanisme d'anticipation	49
3.2.1 Méthode d'anticipation	49
3.2.2 Passage en Mode Dégradé	51
3.3 Architecture Logicielle	53
3.3.1 Task Wrapper Component (TWC)	55
3.3.2 Core Control Component (CCC)	56
3.3.3 Définition des constantes de Contrôle	57
3.4 Application industrielle	57

3.4.1	Spécification fonctionnelle	57
4	Protocole et démarche expérimentale	61
4.1	Principe Général et Objectifs	62
4.2	Protocole de conception d'un cas de test	62
4.2.1	Profil des tâches	64
4.2.2	Profil du cas de test et Chaîne de tâches	66
4.3	Protocole d'Implémentation et Calibration	70
4.3.1	Spécification des paramètres de Contrôle	71
4.3.2	Calibration du Mécanisme de Contrôle	73
4.4	Protocole expérimental global	76
5	Implémentation, Résultats et Analyse	79
5.1	Plateforme de développement	80
5.1.1	Plateforme Matérielle	80
5.1.2	Support Logiciel	82
5.2	Logiciel Applicatif	86
5.2.1	Benchmark MiBench	86
5.2.2	Charge de test	88
5.2.3	Implémentation de la plateforme expérimentale logicielle	89
5.3	Application du Protocole Expérimental à MiBench	93
5.3.1	Phase de Conception	94
5.3.2	Phase de Configuration du mécanisme	97
5.3.3	Phase de Validation et Analyses	102
Conclusion		105
Conclusion	105	
Perspectives	107	
Bibliographie		111
Publications		119
Résumé		121

Liste des Abréviations

Modèle de Tâches

τ_i	Tâche, d'indice i
T_i	Période d'apparition d'une tâche
D_i	Date limite d'échéance à la terminaison d'une tâche
C_i	Temps d'exécution pire cas d'une tâche
L_i	Niveau de criticité d'une tâche
$\tau_{i,j}$	jème occurence d'exécution de la tâche τ_i à sa période T_i
$a_{i,j}$	Moment d'activation du job $\tau_{i,j}$
$s_{i,j}$	Début d'exécution du job $\tau_{i,j}$
$e_{i,j}$	Fin d'exécution (terminaison) du job $\tau_{i,j}$

Modèle de Chaîne de tâches

R_j	Temps de réponse bout-en-bout d'une chaîne de tâches
D_c	Date limite d'échéance d'exécution bout-en-bout d'une chaîne de tâche
$succ(\tau_i)$	Fonction qui permet de trouver le job successeur du job τ_i d'une chaîne de tâches
$succ^p(\tau_i)$	fonction itérative pour trouver le pème successeur du job τ_i
$S(t)$	Etat de la chaîne de tâches à l'instant t
$ET(j, t)$	jème trace d'exécution de la chaîne de tâches, à un instant t
$RT(t)$	Temps de réponse partiel d'une chaîne de tâche active à l'instant t
$rWCRT(t)$	Pire Temps de Réponse Restant d'une chaîne de tâche

Mécanisme de Surveillance et Contrôle

TWC	Task-Wrapper Component : module d'encapsulation pour la surveillance des tâches
CCC	Core Control Component : module de décision de changement de mode
T_{CCC}	Période de fonctionnement du Core Control Component
W_{MAX}	Durée maximale garantie entre 2 points de surveillance
t_{SW}	Latence de changement de mode

Introduction

La complexité des systèmes cyberphysiques s'est accrue dramatiquement ces dernières décennies, tant par la multiplication des besoins applicatifs que par la sophistication des architectures matérielles qui les supportent.

C'est ainsi que le domaine de l'automobile est successivement passé du tout mécanique à des architectures Électrique et Électronique (AEE) de plus en plus sophistiquées. Bien évidemment, cette tendance lourde a permis de rendre aux clients des services plus avancés et pertinents qui ont gagné en intelligence en s'appuyant sur les progrès des techniques numériques. Cela s'est fait tout particulièrement grâce à une approche logicielle prépondérante en délaissant les anciens systèmes mécaniques ou électro-mécaniques.

Ces évolutions progressives dans la voiture ont mené à des ajouts de calculateurs ayant chacun son lot de fonctionnalités avancées, potentiellement accompagnées des capteurs (température de l'habitacle, présence sur les sièges...) mais aussi des actionneurs (système d'air conditionné, vitres, verrouillage centralisé...) nécessaires. C'est de cette façon que l'architecture distribuée dans l'automobile s'est étoffée pour atteindre de 50 jusqu'à 100 calculateurs pour les plus hauts de gamme. À terme, cette approche ne semble plus soutenable au vu de la demande en fonctionnalités supplémentaires liées aux technologies émergentes : le véhicule autonome et connecté.

C'est pour cette raison que la tendance d'ajout de calculateurs à une architecture distribuée toujours plus complexe est en train de s'inverser. L'architecture des systèmes embarqués a progressivement profité de l'émergence de calculateurs multicœurs puissants qui peuvent se substituer à nombre d'ECU (Unité de Commande Électronique) élémentaires. L'architecture actuelle s'oriente donc vers des architectures fédérées mettant en jeu des processeurs sur lesquels la coexistence d'applications critiques et non-critiques devient incontournable.

Ces systèmes à criticité mixte sur calculateurs multicœurs induisent des problèmes d'*interférence* d'exécution entre les tâches, de par le partage de ressources matérielles qui peuvent provoquer des latences d'exécution, quand plusieurs logiciels doivent exploiter une ressource commune. Cela a pour conséquence première d'ajouter des risques d'augmentation de temps d'exécution. Pour les tâches critiques ces augmentations peuvent aller jusqu'à un dépassement des échéances temporelles telles que définies par les spécifications. Ce risque de défaillances temporelles ne peut être pris à la légère et doit par conséquent être traité par des mécanismes de sûreté de fonctionnement dédiés. Ces derniers permettant de minimiser autant qu'il faut l'occurrence de défaillances catastrophiques liées au dépassement d'échéance.

Il existe d'ores-et-déjà un certain nombre de solutions propres à la garantie d'exécution de tâches temps-réel. Ceci étant dit, un compromis est fait avec l'exploitation des ressources matérielles. De façon générale, plus les garanties de sûreté de fonctionnement logicielle sont fortes et plus les calculateurs multicœurs sont exploités de façon sous-optimale pour obtenir ces garanties. Ajouté à ce premier enjeu de garanties temporelles, vient donc un second enjeu complémentaire qui est d'avoir

une approche conservatrice pour exploiter au maximum la puissance de calcul disponible. Cette approche conservatrice se définit comme une utilisation optimisée des ressources de calcul pour les tâches non-critiques autant que possible tout en garantissant les échéances temporelles propres aux tâches critiques.

C'est dans ce cadre que sont présentés ces travaux de thèse, focalisés sur l'étude de tâches logicielles critiques représentées sous forme de chaînes fonctionnelles. De fait chaque fonctionnalité exécutée correspond à une chaîne de différents composants logiciels exécutés séquentiellement. L'idée est d'avoir une vision plus macroscopique de l'exécution des tâches en tant que partie d'un ensemble de chaînes fonctionnelles plutôt que des instances isolées qui s'ordonnent l'une après l'autre sur le processeur. Dans cette optique, on considère possible que les aléas de temps d'exécution puisse se compenser de façon naturelle pendant le fonctionnement. Que certains retards sur le temps d'exécution d'une tâche soient *in fine* sans conséquence grâce au fait que des tâches précédentes faisant partie de la même chaîne fonctionnelle aient bénéficié d'une avance sur leur temps d'exécution. Ou inversement, qu'un léger retard d'exécution d'une tâche à cause d'interférence soit bénin de part une compensation du retard dans la suite de la chaîne de tâches.

Avec ce concept, on souhaite mettre en place un mécanisme qui puisse prendre en compte l'état d'exécution d'une chaîne de tâches critiques de façon dynamique. Cette Surveillance doit vérifier qu'il est possible à chaque instant de garantir l'exécution bout-en-bout de la chaîne sous l'échéance temporelle propre à la fonction réalisée. Pour ce faire, un calcul d'anticipation de risque en pire cas doit s'opérer. L'objectif est d'anticiper le risque d'être dans une situation où un dépassement d'échéance deviendrait inévitable du fait d'un surplus d'interférences inter-tâches. Dans l'éventualité où ce cas de figure est anticipé, alors il faudra mobiliser un mécanisme de Contrôle. Son rôle sera de garantir en toute circonstance le respect de l'échéance et donc éviter toute faute temporelle par la prévention de toute interférence supplémentaire à la source : par la suspension des tâches non-critiques.

Dit autrement, avec cette approche plus fonctionnelle de l'exécution du logiciel, le but est d'obtenir un mécanisme de Surveillance et de Contrôle qui offre des garanties minimales sur une chaîne de tâches critiques, par anticipation de risques d'interférences qui pourraient mener à une défaillance. Le principe fondamental de l'anticipation est de mettre en pause temporaire les tâches non critiques pour neutraliser les interférences associées. L'objectif est d'ainsi obtenir une garantie d'exécution bout-en-bout de la chaîne de tâche sans dépassement d'échéance, avec une limitation des interférences uniquement quand nécessaire. De cette façon, le reste du temps où cela n'est pas nécessaire, toutes les tâches du système puissent exploiter au maximum la puissance de calcul et les ressources disponibles.

L'étude de cette problématique, l'approche que nous proposons, son développement et son illustration expérimentale seront abordés en 5 chapitres.

Dans le chapitre 1, nous nous attachons à expliciter la problématique de la thèse en débutant par état des lieux du contexte industriel et de ses contraintes dans l'usage de systèmes à criticité mixte sur calculateurs multicœurs.

Dans le chapitre 2, nous faisons référence aux travaux connexes étudiés. Les recherches sur la problématique du temps réel en général présentent un corpus

foisonnant, voire gigantesque, de par la multiplicité des sous-branches du domaine. Nous positionnons notre proposition d'un point de vue plus macroscopique dans cet ensemble pour nous focaliser sur les travaux les plus pertinents relativement à notre approche.

Dans le chapitre 3, nous décrivons l'approche et la méthode propre à notre contribution pour répondre à la problématique soulevée : allier optimisation des ressources de calcul et garanti temps-réel dans un environnement à criticité mixte complexe.

Dans le chapitre 4, nous développons et justifions notre approche expérimentale qui servira de squelette à la caractérisation et l'implémentation de notre approche.

Enfin, le chapitre 5 est dévolu à l'illustration de notre approche par la présentation d'un cas d'implémentation complet où l'on présente la plateforme matérielle et logicielle sélectionnée ainsi que les choix de développement qui ont été effectués. Ce cas d'étude se présente comme une première preuve de concept et nous permet d'analyser les résultats obtenus pour caractériser notre proposition aux travers de plusieurs expérimentations.

CHAPITRE 1

Enjeux des Systèmes Embarqués Complexes

Sommaire

1.1	Évolutions des Systèmes embarqués	6
1.1.1	Nouveaux systèmes intelligents et connectés	6
1.2	Risques et Problématique	11
1.2.1	Sûreté de Fonctionnement Informatique	12
1.2.2	Systèmes temps-réel et Ressources partagées	14
1.2.3	Problématique et Objectifs	17
1.3	Contraintes et Hypothèses	18
1.3.1	Contexte industriel Automobile	18
1.3.2	Standards industriels et Concept de Criticité	19
1.3.3	Contraintes d'intégration	20
1.4	Grandes approches du domaine	22
1.4.1	Mécanismes de contrôle	22
1.4.2	Mécanismes Réactifs	23
1.5	Contribution de la thèse et objectifs	24

La conception des systèmes embarqués, typiquement automobiles, a subi de fortes évolutions orientées vers de nouvelles fonctionnalités centrées sur le logiciel. Ces évolutions demandent des capacités de calcul de plus en plus importantes et donc des architectures matérielles pour supporter la demande grandissante en fonctionnalités. Par ailleurs le contexte industriel mène à la disparition des calculateurs d'antan, monocœurs, pour se focaliser sur des calculateurs plus complexes et puissants, multicœurs. Cette tendance au multicœur provient à la fois d'une limitation technologique et d'un besoin grandissant : la façon d'augmenter les capacités de calculs par les méthodes classiques (montée en fréquence) atteint ses limites et les capacités d'exécution concourante de logiciel est de plus en plus demandée dans un contexte aux contraintes financières et de *time-to-market* fortes.

C'est ainsi que né la volonté de passer sur des architectures électriques et électroniques plus centralisées via l'utilisation d'une quantité réduite d'unités de calcul, mais intégrant un plus grand nombre de fonctionnalités de traitement en leur sein ; en un mot, des processeurs multicœurs. Cette volonté implique cependant une superposition des difficultés inhérentes aux architectures matérielles plus complexes avec les contraintes de sûreté de fonctionnement du logiciel. Nous faisons donc face à des systèmes à criticité mixte exécutés sur des calculateurs aux mécanismes complexes.

Nous verrons ainsi dans ce chapitre quels sont les aspects essentiels de ce contexte et ses spécificités à prendre en compte pour proposer de nouveaux éléments de réponse dans la conception de systèmes à criticité mixte sur processeurs multicœurs. Nous conclurons cette partie avec la présentation de la problématique à laquelle nous tenterons de répondre ainsi qu'une vue d'ensemble des différents chapitres qui structurent cette thèse.

1.1 Évolutions des Systèmes embarqués

1.1.1 Nouveaux systèmes intelligents et connectés

Si l'on prend le cas du domaine automobile, depuis près de trente ans l'industrie n'a cessé de faire évoluer la façon de concevoir les véhicules et notamment leurs systèmes sous-jacents. Comme illustré avec le diagramme 1.1, la transition s'est faite de modifications purement mécaniques vers des évolutions électriques, puis électroniques et de plus en plus intelligentes. Les systèmes de divertissement du consommateur ont été les premiers, dans le milieu des années 1920, à introduire des composants électroniques au sein des véhicules sous la forme de récepteurs radio à lampes ! Si l'apparition de transistors, dans les années 1950, a contribué à l'amélioration des capacités techniques des appareils et à la diffusion massive des autoradios au sein des automobiles, le concept de base a peu évolué jusqu'à la fin des années 1970. L'introduction des premiers systèmes de navigation dans les années 1980, puis des systèmes multimédia dans les années 2000 a changé la donne. Désormais, l'ancienne façade de l'autoradio devient un écran de commande nommée *head unit* et concentre 70% du code du véhicule. Les voitures se sont modernisées avec l'ajout de calculateurs dédiés à des fonctions internes ou des services. Le développement des technologies de l'industrie 4.0 mène à une augmentation exponentielle du logiciel embarqué dans l'automobile au cours des 15 dernières années [Blanchet 2016], avec la présence de plus de 50 calculateurs embarqués dans certains modèles [Juliussen 2022]. Les contrôles mécaniques et autres systèmes électriques "simples" cèdent la place au monde du numérique. Les équipements électroniques et logiciels se multiplient au sein du véhicule pour l'aide à la conduite (*Advanced Driver-Assistance System – ADAS*) et l'ajout de services [Schmidt 2010]. De fait, le système multimédia moderne a un rôle qui va bien au-delà de celui du simple autoradio : il devient l'interaction principale entre le consommateur et le véhicule et devient un critère de choix prépondérant à l'achat.

Ainsi, du simple Système Anti-blocage des roues (ABS), on a introduit des Assistants à la Conduite tels que le Freinage d'Urgence (*Emergency Braking System*) ou encore le Système de Gestion de Ligne (*Lane Support System*) qui permet à la fois l'Avertissement de Dépassement de Ligne (*Lane Departure Warning*), l'Assistant de Maintien de Ligne (*Lane Keeping Assist*) et le Maintien de Ligne d'Urgence (*Emergency Lane Keeping*)... et il ne s'agit là que de 2 fonctionnalités supplémentaires ! En parallèle, la voiture devient de plus en plus automatisée, voire autonome. Elle gagne en connectivité avec la prise en compte de données extérieures possiblement avec un lien direct au cloud pour proposer une diversité de services : météo, divertissement, trafic routier, pour n'en citer que quelques exemples. Les systèmes embarqués

deviennent par conséquent aussi connectés. On parle de communications *car-to-car* entre véhicules ou *car-to-infrastructure* entre véhicule et infrastructures routières par exemple.

Cette ouverture du système à son environnement est à double tranchant. D'une part cela offre de nouveaux horizons de fonctionnalités et optimisations de conduite, avec des possibilités d'évolutivité simplifiée. Mais d'autre part la complexité va grandissante avec les enjeux d'ingénierie que cela implique.

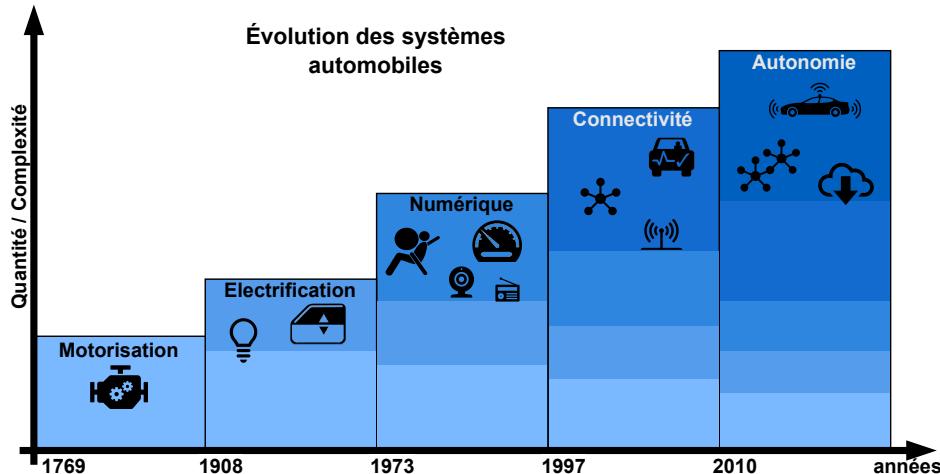


FIGURE 1.1 – Principaux domaines d'évolution des systèmes automobiles au fil du temps

De façon plus générale, le contexte industriel actuel fait émerger de nouvelles technologies basées sur des logiciels de plus en plus complexes et performants. Cela est rendu possible via l'émergence d'architectures matérielles toujours plus puissantes et performantes. Ces améliorations permettent le développement et la mise en application de nouvelles technologies comme les réseaux de communication sans fil haute performance ou encore l'usage d'intelligences artificielles. On retrouve ainsi un nombre grandissant de fonctionnalités directement embarquées dans l'automobile, l'avion, le train pour répondre à la fois à de nouveaux besoins fonctionnels : assistance à la conduite/pilotage, tableaux de bord, etc. et à des besoins de confort d'usage : info-divertissement, connectivité, automatisations...

D'un point de vue logiciel, les mises à jour de systèmes embarqués incluent à la fois de nouvelles fonctionnalités critiques pour le bon fonctionnement du système, mais aussi l'ajout de fonctions moins critiques. Ces mises à jour de services non essentiels amplifient la multiplicité des niveaux de criticité du logiciel embarqué et donc la cohabitation entre sous-systèmes critiques et sous-systèmes non-critiques que l'on pourrait qualifier de "confort".

D'un point de vue matériel, il y a de fortes convergences sur les architectures employées dans les différents domaines. Historiquement, on retrouvait en premier lieu des calculateurs monocœurs. Cependant, les diverses évolutions d'exigences ont fait apparaître des limites en capacité de calcul. La montée en fréquence de fonctionnement atteint un seuil maximum à cause de la chauffe et la consommation que cela

implique. Tandis que l'augmentation du nombre de transistors qui composent les processeurs arrive aux abords des limites physiques : la taille de gravure du silicium arrive au même ordre de grandeur que la taille des atomes de silicium dont elle est composé. De fait, jusqu'à récemment encore, la Loi de Moore [Thompson 2006] sur la puissance des processeurs s'est vérifiée. Des premiers microprocesseurs Intel en 1971, avec quelques milliers de transistors de $10\text{ }\mu\text{m}$, l'on est aujourd'hui à plus de 1 milliards de transistors de près de 10 nm. Mais à l'aune d'une gravure proche des 2 nm, on environne les dimensions de 10 à 15 atomes et les effets de la physique quantique entrent en jeu. Par conséquent, l'on se dirige vers les limites des technologies actuelles pour poursuivre ces améliorations de puissance. Pour ces raisons, le plus grand levier de progression disponible aujourd'hui repose sur la parallélisation des unités de calcul, et donc la notion de calculateur multicœur, qui est apparue dès les années 1950 [Smotherman 2005]. Les fondeurs s'orientent vers des processeurs où la montée en puissance est assurée par la multiplication des unités de calcul (dit "cœurs") parallèles dans le processeur. On passe ainsi de monocœurs toujours plus petits et compacts à des duals/quatrième cœurs... et l'on va aujourd'hui jusqu'à des supercalculateurs à plus de 128 cœurs. Tous ces changements se visualisent parfaitement avec l'évolution des caractéristiques des processeurs au fil des années en Figure 1.2, tel qu'agrégé par K. Rupp [Rupp 2020]. Cette évolution est la bienvenue dans tous les secteurs concernés, allant du grand public dans les ordinateurs, téléphones et autres multimédias jusqu'aux applications industrielles en passant par les usages de serveurs réseaux et centres de calculs.

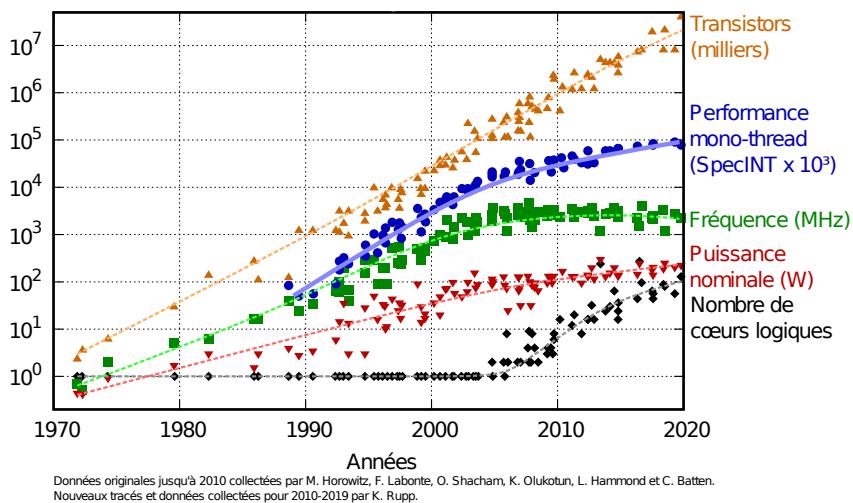


FIGURE 1.2 – 42 Ans d'évolution des processeurs - Tendances

Il existe divers types d'architectures matérielles parmi les évolutions multicœurs que l'on retrouve aujourd'hui. On pourrait de façon simple différencier entre les multicœurs classiques, les manycœurs et à l'extrême ce que l'on connaît sous le nom de GPU, les processeurs graphiques.

Multicœurs "classiques" Les calculateurs multicœurs "classiques" disposent d'un certain nombre d'unités de calculs ("cœurs"), auxquelles sont adjointes diverses zones mémoires (cache, RAM, ROM). Le tout est piloté par des contrôleurs et bus de transfert de données pour interconnecter les cœurs, les cellules mémoires et les entrées/sorties. Dans les versions les plus récentes, des modules dédiés peuvent être ajoutés pour des fonctionnalités spécifiques comme le chiffrement.

Il existe des variations d'architectures que l'on peut notamment distinguer entre d'une part les multicœurs basés sur le cache (comme celui susmentionné) qui sont prédominants et d'autre part les multicœurs basés sur *scratchpad*, c'est-à-dire des mémoires locales dédiées à chaque cœur. On peut voir la différence fondamentale de structure entre ces deux variations sur la Figure 1.3.

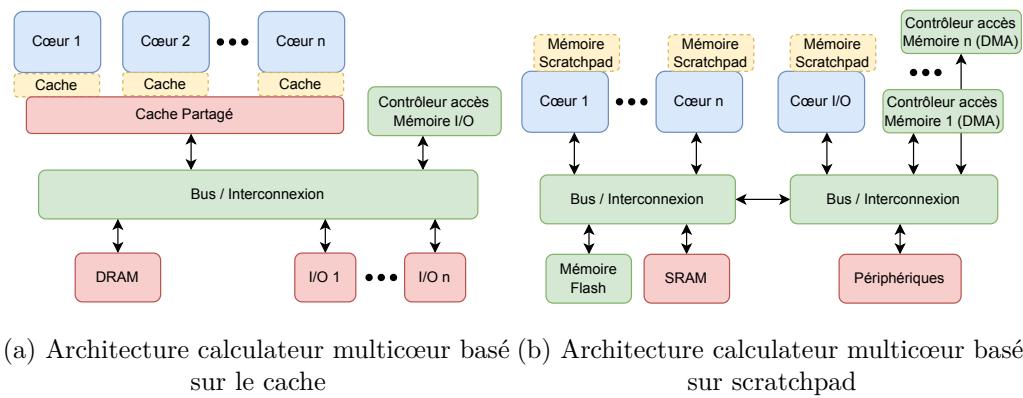


FIGURE 1.3 – Exemple multicœur et mémoires

Les architectures à Scratchpad mettent à disposition des mémoires à haut débit dédiées à chaque cœur. Toutes les ressources sont le plus possible séparées entre les unités de calcul. Les modèles d'exécution de tâche sont en général très cadrés avec trois phases d'exécution (Chargement mémoire, exécution et déchargement) pour maîtriser le modèle d'exécution du logiciel. Les architectures Scratchpad sont alors des solutions en elles-mêmes aux problèmes d'exécution parallèle de code sur les calculateurs multicœurs pour éviter les problèmes d'interférences. Ce type de solutions par design limite en revanche les possibilités d'implémentation de logiciel de part cette spécificité de l'architecture matérielle.

À l'inverse, les architectures basées sur le cache se veulent polyvalentes et comptent sur des mécanismes de contrôle logiciel pour optimiser l'utilisation des ressources. La mémoire est partagée à différents degrés entre les cœurs. De façon à décongestionner les accès mémoire et accélérer ces dernières, une hiérarchie mémoire est mise en place, associant des espaces mémoire progressivement plus petits et rapides en fonction de leur proximité au processeur. Il s'agit ici de trouver un équilibre entre coût de la mémoire et vitesse d'accès aux données. En effet, cette dernière dispose de trois caractéristiques antagonistes :

- la **latence** - temps d'accès aux données,
- la **bande passante** - débit de données accessible,
- la **taille** mémoire - espace mémoire disponible (pour un coût donné).

Un espace mémoire pourra être soit de petite taille, mais rapide au niveau de son temps d'accès, soit de grande taille et plus lent comme schématisé dans la Figure 1.4b. On a par conséquent au plus proche des coeurs les registres, de taille très limitée (octets) mais au temps d'accès très rapide : ils sont la base pour toutes les opérations effectuées par le processeur. À l'opposé, la mémoire principale, de très grande taille (Go/TB) pour laquelle tous les coeurs doivent passer par un bus commun pour y accéder. C'est donc la mémoire la plus lente d'accès, mais aussi la moins coûteuse. Plusieurs intermédiaires ont été mis en place entre ces deux types de mémoire. Il s'agit typiquement de niveaux de cache qui peuvent être non partagés, c'est-à-dire propres à chaque cœur ou bien commun à tous. Le dernier niveau de cache, partagé, est classiquement appelé LLC (*Last Level Cache*) et donne la limite entre les espaces mémoire limités en cache avec des accès rapides d'une part et la mémoire principale qui va provoquer de grands ralentissements d'autre part. On retrouve ainsi avec l'exemple de la Figure 1.4a un cas de calculateur multicœur basé sur le cache, avec 8 coeurs, des niveaux de cache mémoire séparés (L1 et L2) et partagé (L3) ainsi que le bus d'accès à la mémoire principale.

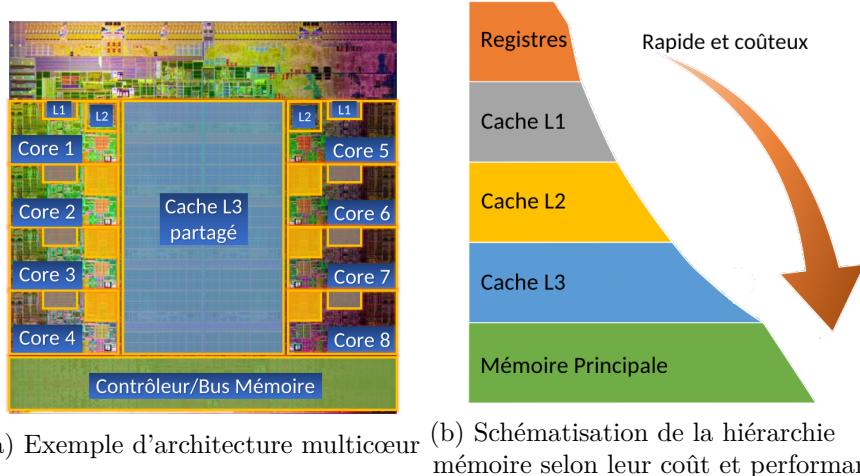


FIGURE 1.4 – Exemple de multicoeur Intel 17-5960X et hiérarchie mémoire

La gestion du contenu de ces caches (en lecture et écriture) est gérée par une politique d'accès mémoire. Cette politique est essentielle à un usage efficace des caches du fait de leur espace limité qui demande à faire des choix sur son usage. Cela est peu documenté par les constructeurs, et chacun aura son propre algorithme. La méthode de base la plus répandue étant empirique, par principe de localité temporelle [Durrieu 2014] et spatiale [Wilkes 1965]. On considère que plus une donnée a été récemment accédée, plus elle a de chance d'être à nouveau utilisée. De même si une donnée est sollicitée, alors les données proches spatialement ont aussi plus de chance d'être utilisées. Nous n'iront pas plus dans les détails sur les politiques de gestion d'accès à la mémoire. Il faut garder à l'esprit qu'elle est plutôt subie par les industriels qui intègrent le matériel dans leurs systèmes. Pour un processeur donné on aura certaines performances de calcul et accès mémoire, et il faudra mettre en comparaison les performances "par défaut" d'un logiciel sur une architecture

matérielle donnée face au même logiciel avec l'ajout de la surcouche de contrôle d'exécution apportées par l'intégrateur pour limiter les interférences.

Calculateurs manycœurs Les calculateurs manycœurs sont des microprocesseurs incluant un grand nombre de cœurs dans l'objectif primaire d'une plus grande capacité d'exécution de code parallèle. Pour ce faire, les cœurs peuvent être spécialisés avec la réduction des instructions réalisables et optimisations à des tâches spécifiques. C'est la différence principale avec les multicœurs qui possèdent en général des cœurs identiques (processeur homogène) avec de bonnes performances à la fois en série et en parallèle. Les architectures manycœurs grâce à leurs spécificités demandent des méthodes de programmation appropriées pour pouvoir être pleinement exploités dans le cadre d'une application. Cela augmente donc le niveau de complexité de développement, mais au bénéfice d'une forte amélioration des performances.

Les GPUs (*Graphic Processing Unit*) sont un cas particulier de manycœurs à présent très répandu pour des usages variés [Owens 2008]. Cette forte expansion des GPU est due non seulement aux capacités de rendu graphique, mais surtout à leurs capacités de programmation parallèle poussée au maximum. Un grand nombre de domaines, notamment dans la recherche, y voient donc un microprocesseur d'usage général à hautes capacités de calcul parallèle. Les GPU sont efficaces du fait qu'ils permettent de réaliser le même calcul sur un très grand nombre de données différentes (typiquement calculs matriciels) pour obtenir tout autant de résultats en sortie. Il s'agit d'un modèle dit *SIMD - Single-Instruction, Multiple-Data*. Là où les multicœurs conventionnels se focalisent sur des cœurs versatiles qui s'adaptent pour pouvoir gérer tous les cas d'applications, les GPU se focalisent sur la réalisation de tâches identiques en parallèles, ils restent donc spécialisés pour des types de tâches spécifiques, en complément de processeurs plus polyvalents.

Dans le cadre de ces recherches, nous nous focaliseront sur le dénominateur commun le plus utilisé dans les architectures électriques et électroniques, qui est donc le processeur multicœur basé sur le cache.

1.2 Risques et Problématique

Dans le cadre du contexte automobile, on se dirige vers un nouveau paradigme, où la voiture n'est plus un système mécanique sur lequel on adjoint du logiciel, mais à l'inverse un superordinateur multifonctionnel auquel on implante des roues et un moteur. Les systèmes automobiles sont ainsi devenus des systèmes cyberphysiques qui entrent en interaction à la fois avec les utilisateurs et l'environnement. On distingue deux grands domaines de logiciels embarqués dans le véhicule. Tout d'abord l'info-divertissement, qui réunit les systèmes multimédias et autres affichages non nécessaires à l'usage primaire du véhicule. Et deuxièmement les calculateurs enfouis qui réalisent des fonctions essentielles qui ne sont pas nécessairement visibles de l'utilisateur, telles que le contrôle moteur. Pour soutenir ces besoins émergents, il est nécessaire de se baser sur des architectures matérielles plus puissantes comme les multicœurs. Cependant, cette disruption apporte de nouveaux enjeux, notamment de

sécurité, vie privée, mais aussi sur la prédictibilité et la sûreté de fonctionnement du système à cause de sa complexification. Cela fait donc évoluer les systèmes embarqués dans un environnement profondément à risques, mais qui en plus s'accompagne de contraintes fortes. Nous nous devons donc d'introduire ici les notions de Sûreté de fonctionnement nécessaire à l'analyse.

1.2.1 Sûreté de Fonctionnement Informatique

La sûreté de fonctionnement (SdF) d'un système informatique est "*la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre, le service étant le comportement du système perçu par un utilisateur, cet utilisateur étant un système (informatique, humain, environnemental) qui interagit avec le premier.*" [Laprie 1996]. C'est donc la capacité d'un système informatique de répondre de manière correcte, conformément aux spécifications fonctionnelles, à une requête d'un autre système. La sûreté de fonctionnement est définie en fonction de trois notions principales : **a)** les *attributs* qui définissent les propriétés assurées, **b)** les *entraves* qui caractérisent les circonstances indésirables mais prévues, **c)** et les *moyens* qui précisent les techniques permettant au système de fournir son service. Selon les services souhaités par l'utilisateur, ce dernier peut vouloir accentuer certaines propriétés pour assurer le bon fonctionnement du système. Ainsi la sûreté de fonctionnement englobe les attributs suivants :

- La **disponibilité** - la capacité d'être prêt à délivrer le service correct ;
- La **fiabilité** - l'assurance de continuité d'un service correct ;
- La **sécurité-innocuité** - l'assurance de non-propagation de conséquences catastrophiques à l'utilisateur ou l'environnement ;
- L'**intégrité** - l'assurance de non-altération du système ;
- La **maintenabilité** - l'aptitude à la réparation et à l'évolution du système.

Ces attributs permettent d'une part d'exprimer les propriétés devant être respectées par le système, et d'autre part d'évaluer la qualité du service délivré vis-à-vis de ces propriétés. Les aspects de sécurité, au sens de la confidentialité et des attaques face à des actions malveillantes indésirables ainsi que la confidentialité, c'est-à-dire, la non-divulgation d'information non autorisée, ne seront pas abordés dans cette thèse.

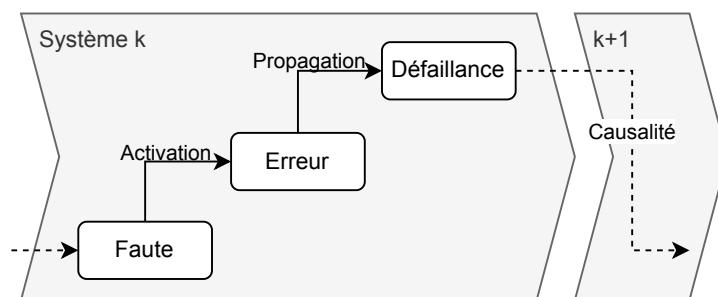


FIGURE 1.5 – Sûreté de fonctionnement - chaîne de causalité

Les entraves à la sûreté de fonctionnement sont les défaillances, les erreurs et les fautes. Une défaillance est une transition d'un service correct vers un service incorrect. Un service est considéré incorrect s'il n'est pas conforme à la spécification ou si la spécification ne décrit pas avec précision la fonction du système. Étant donné qu'un service consiste en une séquence d'états externes du système (observés par l'utilisateur), la survenue d'une défaillance signifie qu'au moins un des états externes s'écarte de l'état correct du service. La déviation est liée à une erreur, qui représente la partie de l'état interne du système pouvant entraîner une défaillance, dans le cas où elle atteint l'interface du service du système. La cause déterminée ou présumée d'une erreur est appelée une faute. La Figure 1.5 représente ce lien de cause à effet. Le fait de prévenir la causalité entre fautes est défaillances pour le bon fonctionnement se désigne par la méthode de silence sur défaillance. C'est-à-dire qu'une faute ou une erreur n'aura pas plus de conséquences et ne provoquera pas autre de défaillance, ou inversement.

Pour minimiser l'impact de ces entraves, la sûreté de fonctionnement dispose de méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement des fonctions du système. Le développement d'un système sûr de fonctionnement passe donc par l'utilisation combinée de ces méthodes, appelés moyens, pouvant être classées en quatre types :

- **Prévision** des fautes : estimation de la présence, de la création et des conséquences des fautes (p. ex. Analyse FMEA) ;
- **Prévention** des fautes : méthodes visant à réduire les occurrences ou l'introduction de fautes (p. ex. outil de génie logiciel, processus de développement strict) ;
- **Élimination** des fautes : réduction du nombre et de la sévérité des fautes (p. ex. test, injection de fautes) ;
- **Tolérance** aux fautes : capacité de fournir un service, optimal ou dégradé, en présence de fautes (p. ex. techniques de redondance).

La prévention et la tolérance aux fautes visent à fournir la capacité de délivrer un service correct, tandis que l'élimination et la prévision des fautes visent à susciter la confiance en cette capacité en justifiant que les spécifications fonctionnelles de sûreté de fonctionnement et de sécurité sont adéquates et que le système est conforme. Toutes ces techniques sont dédiées à garantir des propriétés de sûreté de fonctionnement issues de spécification non fonctionnelles.

Tolérance aux Fautes Les fautes auxquelles un système doit faire face sont nombreuses et peuvent ne pas avoir d'impact sur celui-ci tant qu'un ou plusieurs événements ne se sont pas produits. On les appelle alors des fautes dormantes. Une fois activées, ces fautes peuvent avoir un impact catastrophique sur le système. D'origines diverses et variées, certaines fautes sont dues à l'environnement, au matériel, ou encore à l'être humain.

Chaque faute peut provoquer une ou des erreurs différentes pouvant entraîner la défaillance du système. Malgré l'application des techniques de prévention et d'élimination des fautes, certaines subsistent et sont à même d'être activées.

Un système tolérant aux fautes doit pouvoir assurer à l'utilisateur un service correct en dépit des fautes pouvant altérer ses composants, durant sa conception ou son interaction avec d'autres systèmes [Avizienis 2004]. La Tolérance aux fautes est mise en œuvre grâce aux moyens de **détection** d'erreurs, c.-à-d., l'identification des déviations du service correct, et de **recouvrement**, c.-à-d., les techniques permettant en cas d'erreur détectée de passer d'un état de système fautif à un état assurant un service nominal ou dégradé.

La détection d'erreur peut être soit concurrente et se déroulant pendant l'exécution du système soit anticipée en vérifiant les paramètres du système lors de la suspension de son exécution. Une fois cette erreur détectée, les techniques de recouvrement peuvent être employées, d'une part pour assurer le service désiré et éviter la propagation de l'erreur (traitement des erreurs) et d'autre part pour isoler le composant fautif, diagnostiquer l'erreur, trouver et déterminer la faute originelle pour assurer une opération de maintenance (traitement des fautes).

Les techniques de détection et de recouvrement sont nombreuses et sont regroupées dans des mécanismes de tolérance aux fautes associés à un ou plusieurs types de fautes. Il n'y a à l'heure actuelle aucun mécanisme générique pouvant pallier n'importe quel type de fautes ou d'erreurs. Que cela soit de la redondance matérielle, logicielle, temporelle, de la diversité dans l'implémentation ou l'architecture, les techniques sont nombreuses et souvent propres à chaque domaine et au budget alloué à la tolérance aux fautes.

Dans le cadre de ces travaux de recherche, nous nous intéresseront particulièrement à la tolérance aux fautes qui a trait donc à la bonne exécution de tâches hébergées au sein d'un même calculateur. Dans le contexte industriel susmentionné, un même calculateur exécute des tâches pour des fonctionnalités variées et par conséquent avec des niveaux de criticité variés. Cela engendre notamment des contraintes sur les temps d'exécution des logiciels les plus critiques. C'est ce qu'on qualifie de systèmes temps réel. Nous sommes en résumé dans un contexte à criticité mixte, où du logiciel de système temps-réel va coexister avec du logiciel avec des contraintes temporales moins strictes, voire aucune contrainte. L'implémentation de mécanismes de sûreté de fonctionnement dans ce contexte-là relève alors de la gestion de fautes temporales dans un système à criticité mixte.

1.2.2 Systèmes temps-réel et Ressources partagées

Système temps-réel Les systèmes embarqués sont conçus sur la base d'un modèle de capteurs et actionneurs. Les capteurs représentent l'ensemble des éléments qui permettent d'obtenir les données d'entrée au système de façon à ce qu'il puisse réaliser sa fonction. Il s'agit notamment des informations de l'environnement du véhicule, mais aussi des données internes avec tout l'état de fonctionnement actuel ainsi que les interactions avec l'utilisateur. Ces informations sont alors gérées par les calculateurs de décision via des algorithmes plus ou moins complexes. Le logiciel permet donc à partir de ces données d'entrée de calculer les commandes qui sont dirigées vers les actionneurs. Les actionneurs sont alors en bout de chaîne afin d'accomplir la commande. Dans le cas où les données d'entrée fournies par les capteurs sont liées aux données de sortie, on parle alors d'une *boucle* de contrôle.

Typiquement avec le chauffage d'un logement qui utilise un capteur de température pour une consigne de température donnée.

Prenons un exemple hypothétique de contrôle de l'injection moteur pour une voiture. En entrée, le calculateur de contrôle moteur récupère entre autre les informations du capteur de vitesse de rotation du moteur, de la quantité d'essence en réservoir et l'accélération demandée par le conducteur. Il peut alors calculer l'instant et la quantité de carburant qu'il sera nécessaire d'injecter dans le moteur. Cette commande est alors transmise à l'actionneur, l'injecteur, pour être réalisée. Et ce bloc de contrôle-commande doit se répéter périodiquement pour suivre la consigne tout le long de l'utilisation du véhicule.

Tous ces éléments de contrôle-commande ont en commun d'avoir des contraintes temporelles. Le temps de réaction –qui définit la durée entre la récupération des données des capteurs jusqu'à la réalisation de la commande par les actionneurs– peut alors être une donnée critique pour certaines applications comme l'exemple donné ci-dessus (*inutile de dire qu'un contrôle d'injection moteur qui prend trop de temps à déterminer combien de carburant injecter aura des conséquences bien évidemment indésirables...*). Ainsi, ce genre d'applications nécessite à la fois de retourner des résultats corrects, mais aussi de les délivrer dans les temps.

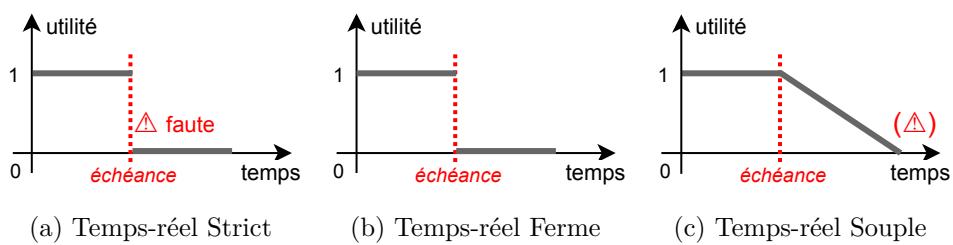


FIGURE 1.6 – Modèles d'utilité des résultats d'une tâche temps-réel

Plus généralement, les applications embarquées se caractérisent par un ensemble de tâches logicielles qui interagissent entre-elles. Elles sont soit périodiques (c.-à-d. exécutés à intervalle réguliers) soit apériodique (sur réception d'un événement). Chaque tâche possède ses spécifications propres en termes de données d'entrée, de sortie ainsi que ses paramètres d'exécution (selon les cas : période, niveau de priorité, allocation physique) dont une échéance d'exécution. Les systèmes temps-réel peuvent se catégoriser en 3 catégories qui sont schématisées en Figure 1.6. On retrouve d'une part les systèmes **temps-réel strict** ("hard real-time") où le respect de l'échéance est strict en Figure 1.6a. Il est alors considéré qu'une tâche dont le temps de réponse dépasserait l'échéance serait une faute temporelle indésirable et les données renvoyées par la tâche n'ont plus de valeur. Le même modèle mais sans conséquences après dépassement de l'échéance est nommé **temps-réel ferme** ("firm real-time") illustré en Figure 1.6b. À l'inverse, les systèmes **non-temps-réel** n'imposent pas de contraintes d'échéance sur l'exécution des tâches. Il s'agit donc de faire au mieux, mais tout dépassement des temps d'exécution nominaux n'a pas de répercussions. C'est ce que l'on côtoie couramment via nos appareils de tous les jours comme le smartphone ou l'ordinateur. Enfin, les systèmes **temps-réel souple** ("soft real-time") sont un entre-deux où l'échéance représente un seuil limite

au-delà duquel la valeur de retour de la tâche garde une utilité pour le système mais qui décroît avec le temps, jusqu'à ne plus être pertinente comme représenté en Figure 1.6c. On dit alors que la donnée est "périmée". Ce dépassement peut alors provoquer ou non une faute.

Les analyses d'exécution temporelles des tâches constituent alors un aspect essentiel du développement de logiciel critique afin de garantir le respect des échéances. Cela peut se faire soit de façon expérimentale ou théorique. L'objectif étant de vérifier l'ordonnancement, c'est-à-dire la bonne gestion de l'exécution du logiciel sur le processeur suivant les contraintes imposées (échéances, dates d'activation, périodes...). Un système est dit prédictible si l'on est capable de prouver de façon théorique que les contraintes temporelles seront respectées. Cela se fait par le biais d'analyses d'ordonnançabilité. Une technique classique de ce type d'analyse consiste à évaluer les pires temps d'exécution ("Worst-Case Execution Time – WCET"). Le WCET indique la durée maximum au-delà de laquelle on sait qu'en toutes conditions, la tâche correspondante aura terminé son exécution. Il est possible de comparer les WCET des tâches avec leurs échéances. Pour du temps réel strict, les valeurs de WCET se devront d'être strictement inférieures aux échéances, là où pour du temps réel ferme ou souple on pourra se contenter d'estimation ou de résultats statistiques.

Au sein d'un même processeur, toutes les tâches n'auront potentiellement pas les mêmes types de contraintes d'exécution. Mais en plus de cela, les architectures matérielles multicoeurs complexifient d'autant plus l'analyse.

Ressources Partagées Comme nous l'avons vu précédemment sur l'architecture multicœur, il existe un bon nombre de ressources qui sont partagées entre les différents logiciels qui sont exécutés. Ces partages de ressources peuvent influencer directement sur l'exécution des tâches et donc sur leur capacité à respecter les échéances. En effet, si plusieurs tâches ont des besoins concurrents d'accès à une même ressource alors nécessairement l'une va passer avant l'autre. Cette dernière sera *de facto* retardée dans son exécution. Il existe ainsi de nombreuses sources de retards potentiels d'exécution [Kotaba 2013] :

— Mémoire –

- erreurs en lecture : si une donnée n'est plus présente en cache du fait qu'elle a été remplacée par les données d'une autre application. Cela demande alors à remonter sur les niveaux de mémoire supérieurs, ce qui engendre des temps d'accès supplémentaires importants ;
- accès concurrents : l'accès concurrent à un niveau de mémoire partagée se fait par le bien d'un contrôleur d'accès mémoire, qui va devoir arbitrer sur l'ordre et le temps alloué à chaque tâche.
- Cohérence mémoire : selon les technologies de gestion de cache utilisées, il faut gérer la cohérence mémoire. Si une donnée est utilisée dans plusieurs mémoires non partagées, alors il faut s'assurer que la donnée en question reste cohérente entre toutes les tâches qui s'en servent. Cela implique en général une synchronisation sur les niveaux de mémoire supérieure quand elle est modifiée de façon locale, et inversement une propagation des modifications vers les tâches qui manipulent la donnée.

- Périphérique et I/O en général – chaque périphérique dispose de son propre contrôleur d'accès. On a donc les mêmes enjeux qu'avec les accès concurrents à la mémoire dans le cas où plusieurs tâches utilisent la même entrée/sortie. Le cas principal ici pour une architecture embarquée est sur l'utilisation d'un bus de communication externe qui sert à interconnecter les calculateurs. L'envoi et la lecture de message sur de tels bus de communication peuvent alors engendrer un grand nombre d'usages concurrents.
- Bus d'interconnexion – Le fonctionnement même des processeurs implique l'utilisation de bus internes afin de gérer la transmission et le stockage de données. Tout usage de ces bus d'interconnexion peuvent alors impliquer des usages concurrents qui impactent l'accès aux données des tâches.
- Puissance de calcul – Enfin, mais pas des moindres, il n'y aura probablement jamais autant de coeurs que de tâches sur un processeur multicœur. Il est de ce fait évident que les tâches devront se partager tout ou partie des coeurs selon leur allocation physique. C'est là que va entrer en jeu la stratégie d'ordonnancement des tâches. La politique d'ordonnancement joue un rôle essentiel pour permettre le respect des contraintes temporelles et optimiser l'usage de la puissance de calcul pour limiter au maximum les temps d'attente en file d'exécution des tâches.

Par conséquent, les contraintes majeures de prédictibilité d'exécution au sein de calculateurs multicœurs est mise à mal. La multiplication des points de contention rendent les analyses de pire temps d'exécution plus complexe. C'est d'autant plus le cas avec la prise en compte de dépendances logicielles (synchronisations) et matérielles (gestion des accès concurrents à des ressources partagées). Cela requiert un bon nombre de compromis sur l'exploitation des ressources matérielles pour compenser cette complexité et conserver les objectifs de prédictibilité d'exécution habituellement recherchés.

1.2.3 Problématique et Objectifs

Dans le contexte industriel qui concerne notre étude, les évolutions des systèmes cyberphysiques présentés précédemment impliquent que des tâches de différents modèles d'exécution doivent être intégrées au sein d'un même multicoeur. On parle alors de système à criticité mixte. Cette coexistence de fonctionnalités va augmenter la complexité d'étude de sûreté de fonctionnement afin de garantir l'ordonnancabilité des tâches et le respect des contraintes temporelles. Et comme nous venons de le voir, les nouveaux calculateurs multicœurs ajoutent en niveau de complexité avec l'augmentation de risque d'interférence entre logiciels concurrents. Il devient par conséquent de plus en plus complexe de mener des études théoriques pour estimer les pires temps d'exécution et donc l'ordonnancabilité des tâches. La conséquence directe à cela est un manque de garanties claires sur le bon respect des échéances temporelles pour les tâches les plus critiques pour lesquelles on ne peut se permettre de telles fautes.

On verra qu'il existe de nombreuses méthodes qui permettent de réduire les interférences et donc fiabiliser les études d'ordonnancabilité. Cependant, cela se fait

en général au prix d'un compromis sur les performances de calcul. Hors, c'est pour cette même puissance de calcul que la transition vers des calculateurs multicœurs s'est faite. Il semble alors essentiel de vouloir l'exploiter au maximum. On a deux objectifs qui s'opposent, mais qui sont tout autant essentiels. D'une part l'exploitation au maximum des capacités de calcul pour héberger tout le logiciel nécessaire aux nouvelles fonctionnalités des systèmes embarqués. D'autre part continuer à donner des garanties fortes de respect des contraintes temps réel pour les tâches critiques.

Cela nous mène donc à la problématique centrale de cette thèse, qui est d'identifier les leviers et mécanismes qui peuvent permettre d'atteindre au mieux les deux objectifs susmentionnés d'optimisation de l'usage du processeur avec les garanties temporelles liées aux systèmes critiques. Nous tenterons dans la suite de proposer une réponse à cette problématique par le biais d'une nouvelle approche qui mène à l'usage d'un mécanisme de surveillance et de contrôle de l'exécution des tâches pour éviter toute faute temporelle en cas d'occurrence d'interférences tout en permettant par ailleurs de libérer toute la puissance de calcul disponible dans l'exécution des tâches.

1.3 Contraintes et Hypothèses

1.3.1 Contexte industriel Automobile

Cette problématique s'inscrit dans un contexte industriel aux contraintes spécifiques. Il est donc important d'avoir ces éléments en ligne de compte pour proposer une analyse et des contributions pertinentes. Historiquement dans le domaine automobile, les calculateurs embarqués étaient conçus de manière *ad hoc*. Le logiciel et le matériel étaient intimement liés. Cela conduit à un nombre de calculateurs très important, chaque calculateur apportant une fonctionnalité qui lui est propre. Les architectures se composent alors d'un grand nombre d'unités de calcul interconnectées.

Ce type d'architecture distribuée présente des inconvénients évidents en terme d'évolutivité du système et de coût de développement. À chaque changement de support physique le logiciel doit passer par un nouveau stade de développement plus ou moins conséquent. Inversement, une mise à jour du logiciel ou un ajout de fonctionnalité demande une prise en compte de l'intégration matérielle avec potentiellement des modifications matérielles pour suivre les évolutions. Chaque ajout de fonctionnalités va de cette façon ajouter de nouveaux calculateurs dédiés, complexifiant d'autant plus l'architecture.

Toutes ces contraintes de développement s'inscrivent dans un contexte bien cadré par des normes et standards. L'architecture fédérée telle qu'elle arrive dans les architectures électriques et électroniques abolit la séparation physique qui préexistait entre les composants logiciels, par leur agrégation dans un nombre réduit de calculateurs plus puissants. Cela résulte en un accroissement de la complexité de l'intégration et de la mise en œuvre de la sûreté de fonctionnement.

1.3.2 Standards industriels et Concept de Criticité

Les processus de développement de systèmes embarqués sont régis par des standards qui donnent des garanties sur le bon fonctionnement et donc vis-à-vis du respect des contraintes non fonctionnelles. Ces standards recommandent des directives de développement qui suivent toutes la durée du processus, de la spécification jusqu'aux tests de validation et d'intégration. Une des normes "mères" de la sûreté de fonctionnement des architectures électriques et électroniques est IEC-61508 [IEC 61508 2010]. Il s'agit d'un standard européen générique qui touche donc à de nombreux domaines tels que le ferroviaire, l'automobile, l'aéronautique, etc. Il s'agit en particulier des systèmes où il existe des risques pour les personnes ou sur l'environnement en cas de défaillances. Ce standard définit des niveaux de criticité SIL (*Safety Integrity Level*). Cela fournit des niveaux de fiabilité requis pour chaque niveau de criticité ainsi que les méthodes applicables pour atteindre cet objectif. Ce standard a été par la suite décliné suivant les domaines. Dans l'automobile est ainsi apparue en 2011 la première version d'ISO 26262, "Véhicules Routiers - Sécurité fonctionnelle [ISO TC22/SC3/WG16 2011].

La norme ISO 26262 est la norme de référence pour la sûreté de fonctionnement dans le domaine automobile. Elle recommande des méthodes et mécanismes, applicables durant toutes les phases de développement du véhicule, pour atteindre et justifier son niveau de sûreté de fonctionnement. La norme préconise d'effectuer une phase d'analyse des risques pour identifier les situations dangereuses et les classifier en 4 niveaux de criticités nommés ASIL (*Automotive Safety Integrity Level*) allant du moins critique (ASIL A) au plus critique (ASIL D). Pour la détermination de ces niveaux, trois critères sont pris en compte : la sévérité, la probabilité d'accomplissement et la contrôlabilité.

- **La Sévérité**

Les conséquences en cas de défaillance peuvent être Légères et Modérées (*S1*), Sévères et potentiellement mortelles – mais survie probable – (*S2*), Potentiellement mortelles -survie incertaine- voire mortelles (*S3*).

- **La Probabilité**

Le risque d'occurrence peut être Très Faible (*E1*), Faible (*E2*), de probabilité Moyenne (*E3*) ou de Haute probabilité (*E4*).

- **La Contrôlabilité**

Les capacités de l'utilisateur à gérer la défaillance. Une défaillance peut être facilement contrôlable (*C1*), normalement contrôlable (*C2*), difficilement, voire impossible à contrôler (*C3*).

Bien entendu, l'interprétation de ces trois critères doit se faire au regard du système étudié, et non de façon absolue et déterministe. Ces critères donnent lieu à une table de classification qui définit alors le niveau d'ASIL des composants tel que décrit dans le Tableau 1.1. On remarquera le niveau "QM" pour *Quality Management*, qui correspond aux composants qui n'impliquent pas de criticité particulière et peuvent alors être développés "au mieux" sans contrainte spécifique. Il est bien entendu moins critique qu'un ASIL A. Avec cette table on connaît alors le niveau de confiance que l'on va imposer à chaque composant pour qu'il puisse être utilisé

de façon sûre de fonctionnement. Plus un composant sera critique, plus son niveau d'ASIL sera élevé en conséquence, et donc plus il faudra apporter d'efforts pour qu'il respecte les standards.

On peut mentionner de la même manière d'autres standards équivalents dans l'avionique, DO-176 où les niveaux de criticité sont désignés en DAL ("Design Assurance Level"), ou encore le ferroviaire avec CENELEC 5012x, mais aussi dans le nucléaire, le spatial, l'automatique, le médical... La plupart dérivés de IEC-61508. Un certain nombre de ces standards ont été comparés dans les travaux de [Baufreton 2010]. On y retrouve un point commun qui nous intéresse tout particulièrement ici qui est sur les contraintes temporelles.

TABLE 1.1 – Matrice de Définition des Niveaux d'ASIL - ISO 26262

		Contrôlabilité								
		C1			C2			C3		
Sévérité		S1	S2	S3	S1	S2	S3	S1	S2	S3
Probabilité	E1	QM	QM	QM	QM	QM	QM	QM	QM	A
	E2	QM	QM	QM	QM	QM	A	QM	A	B
	E3	QM	QM	A	QM	A	B	A	B	C
	E4	QM	A	B	A	B	C	B	C	D

Que ce soit IEC-61508 ou plus spécifiquement ISO 26262, il est clairement stipulé que "*les contraintes temporelles des fonctions à durée critique doivent être gérées par les spécifications de sûreté de fonctionnement logicielle. Ici, à la fois les pires temps d'exécution au niveau du code et les temps de réponse au niveau système doivent être considérés.*". Et précisément, "*l'absence de tout interférence se doit d'être assuré et, tout comme dans IEC-61508, le logiciel est sujet au plus haut niveau d'ASIL impliquée quand l'indépendance temporelle entre les fonctionnalités ne peut être assurée.*" [ISO 26262 Sec. 7 2018]. C'est au regard de ce type de contrainte industrielle qu'il est essentiel de proposer de nouvelles solutions avec l'arrivée de calculateurs multicœurs qui complexifient grandement les garanties de non-interférence.

1.3.3 Contraintes d'intégration

De façon plus générale, les enjeux industriels peuvent varier selon les domaines. Ceci étant dit on peut nommer des points principaux, qui sont ceux que l'on va tenter de prendre en considération dans cette étude. La première d'entre elle est l'imposition de capacités de déploiement rapides ("Time-to-market" réduit). Les itérations entre générations demandent des coûts de développement les moins importants possibles. Cela permet des cycles courts et réactifs qui s'adaptent aux évolutions technologiques. Cette contrainte industrielle est structurante sur les choix de conception, ce qui nous ramène souvent au principe "KISS" pour "Keep It Safe and Simple" dans notre cas. C'est une philosophie que j'ai souhaité maintenir au long de cette thèse afin de tenter une approche un peu différente des principales recherches actuelles qui tentent souvent d'aller dans des niveaux de détails toujours plus précis et complexes pour

répondre aux difficultés technologiques, au détriment d'une facilité d'implémentation qui permettrait une appropriation industrielle. Comme on le verra plus tard, il existe ainsi des solutions très sophistiquées qui donnent de bons résultats théoriques, mais qui ne se sont pas généralisées. Les questions de complexité d'implémentation et simplicité de maintenance dans un cas réel semblent donc relativement déterminantes pour mesurer la pertinence d'une nouvelle contribution à la sûreté des systèmes embarqués.

En ce sens, il existe globalement 3 types de sous-systèmes qui sont intégrés par les constructeurs. D'abord les systèmes en *black box* ou "boîtes noires" qui sont entièrement conçus par un équipementier tiers à partir de spécifications. Le contenu de ces "boîtes" est alors inconnu dans ses détails. Ensuite les systèmes en *white box* où à l'inverse, la totalité de sa conception et de son modèle est connue. Cela permet des tests bien plus en profondeur et donc une plus grande confiance en son comportement. Pour être des "boîtes blanches" les composants sont en général soit directement faits par l'intégrateur final, soit fourni en open-source (plus rare, tristement). Enfin, il y a tout l'entre-deux de composants contenant à la fois des éléments en "boîte noire" et en "boîte blanche", que l'on appelle naturellement "boîtes grises". Il est très important de noter par ailleurs une catégorie transverse des systèmes *Legacy*, qui sont des composants où les altérations et modifications sont impossibles pour diverses raisons. Soit parce qu'il s'agit d'une black box que l'on ne souhaite ou ne peut remplacer, soit pour des raisons de compatibilité restreinte où toute modification risquerait de compromettre la totalité du système intégré. Ces différents types de sous-systèmes sont importants à présenter, car tout ce qui va toucher aux boîtes noires et aux composants *legacy* ajoute une contrainte forte sur le développement de solutions à la sûreté de fonctionnement. De fait, tout mécanisme de sûreté de fonctionnement impliquant la modification ou l'adaptation du code de fonctions intégrées au système risque d'être éliminé d'office !

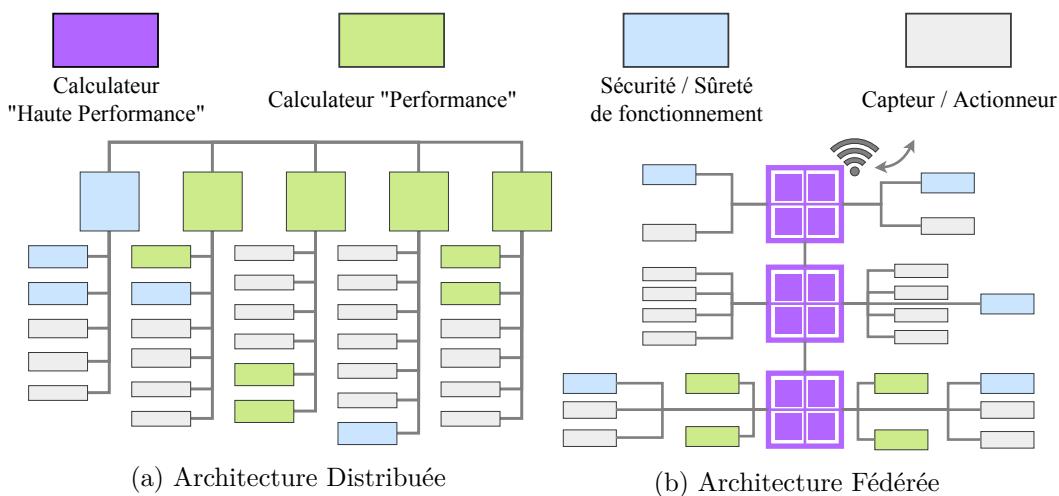


FIGURE 1.7 – Architectures Electrique/Electronique (AUTOSAR)

On pourra pour finir, mentionner des enjeux plus divers tel que les contraintes d'encombrement. Les systèmes embarqués ont une forte tendance à la miniaturisation.

sation pour des raisons diverses selon les domaines. Cela permet une réduction de poids, essentiel pour tous les systèmes volants (avions, drones...) mais aussi d'encombrement pour des domaines comme l'automobile ou le ferroviaire qui doivent en toute circonstance rester dans des dimensions standards. Cette contrainte se fait beaucoup sentir avec l'arrivée des voitures autonomes par exemple, où les premiers prototypes – bien que fonctionnels – se sont avérés trop chargés et encombrants avec le surplus d'équipement pour être transposables facilement en produits commercialisables tel-quel.

Au regard de ces enjeux, l'évolution future naturelle est de réduire le nombre de calculateurs embarqués, en passant d'un grand nombre d'unités de calcul à une quantité limitée de "supercalculateurs", qui vont agréger différentes tâches. On passe de cette façon d'un système distribué à un système fédéré basé sur des calculateurs primaires accompagnés de processeurs satellites qui gèrent le strict nécessaire à hauteur des différents capteurs/actionneurs. Cette différence d'architecture est illustrée en Figure 1.7. Cela permet de réduire les coûts et l'encombrement, qui va diminuer par la même occasion la quantité de câblages requis. Ce type d'architecture va faciliter l'évolutivité qui sera donc bien plus axée sur des mises à jour logicielles sans toucher au matériel. La connectivité permet de mettre en œuvre le concept du véhicule "*as-a-service*", qui va pouvoir évoluer et se mettre à jour régulièrement à distance (*Over-the-Air Updates*).

1.4 Grandes approches du domaine

Nous verrons plus en détail dans le chapitre 2 les différentes solutions actuelles qu'il existe dans le domaine pour répondre à ces problématiques. Nous pouvons tout de même d'ores-et-déjà présenter fondamentalement sur quoi reposent les principes existants afin de mieux situer notre axe de recherche.

La problématique principale à laquelle nous devons faire face réside dans la gestion des interférences matérielles de façon à éviter des fautes temporelles ou tout du moins à couper la chaîne de causalité de façon à ce qu'il y ait toujours silence sur défaillance et donc que le système puisse continuer à fonctionner correctement. Il est possible de différencier deux grands domaines d'approches. D'une part les stratégies de **contrôle** qui sont plutôt statiques et définis hors-ligne lors du développement ; d'autre part les stratégies **réactives** qui sont plutôt dynamiques et évoluent en ligne pendant le fonctionnement.

1.4.1 Mécanismes de contrôle

Ce genre de stratégies consistent à déployer des mécanismes qui limitent les interférences et donc les risques de faute de façon préventive. Le développement et l'implémentation sont alors réalisés d'une manière à ce que par construction, les risquent soit *de facto* rendus impossibles. Cette neutralisation des interférences par construction permet au système d'être *composable*. La composabilité définit sa propension à changer de comportement temporel et fonctionnel selon l'ajout ou le retrait d'autres éléments du système. En d'autres termes, un système est composable s'il est possible de lui ajouter des applications supplémentaires à exécuter sans que

ça modifie le comportement temporel et fonctionnel tel qu'il était précédemment. Ce type de stratégies permet en plus de limiter plus efficacement les explosions de pire temps d'exécution notamment, et donc conserver une exécution du logiciel bien cadrée et maîtrisée pour en contrôler les risques inhérents au matériel.

On peut citer parmi ce genre de techniques :

- Politiques strictes de gestion d'accès aux ressources partagée, avec des intervalles de temps fixes dédiées notamment. Chaque application ayant sa fenêtre temporelle dédiée pour accéder à la ressource partagée, les délais deviennent connus et maîtrisés.
- Séparation temporelle d'exécution des applications : il est possible d'ordonnancer les différentes applications de façon complètement séparées les unes des autres. Cela revient à limiter fortement l'exécution parallèle de code, mais par la même prévient radicalement tout risque d'interférence avec le logiciel ainsi isolé.
- Séparation spatiale des applications : l'allocation d'un espace mémoire dédié pour chaque application permet de résERVER et donc séparer physiquement les applications entre-elles. De cette façon, tout risque de recouvrement des données (c.f. erreurs de lecture) est empêché entre applications qui ont des réservations d'espace mémoire disjoint.

L'avantage du principe de composabilité qui permet l'ajout sans influence de nouvelles fonctionnalités se fait au détriment d'autres caractéristiques importantes. De fait, la plupart des méthodes qui rentrent dans cette catégorie ont en revanche un défaut commun qui est de limiter les capacités d'utilisation du matériel. En effet, on comprend naturellement que si l'on limite la taille mémoire qu'une application donnée est autorisée à utiliser pour son fonctionnement, ou encore si l'on constraint sa plage temporelle d'accès à certaines ressources alors les performances de l'ensemble seront forcément moindre que s'il n'y avait pas ces limitations. Des garanties réduites sur les pires temps d'exécution se font donc au détriment de l'optimisation d'utilisation des ressources matérielles. Aussi, la suppression totale des interférences reste un objectif très complexe qui n'est atteint que partiellement. Les architectures opérationnelles fédérées présentes dans l'automobile par exemple se présentent comme respectant le principe de composabilité avec un calculateur dédié pour chaque prestation du véhicule. Mais il demeure la ressource partagée via le réseau d'interconnexion de tous ces calculateurs. Aussi, les composants provenant de fournisseurs multiples, testés individuellement pour être intégrés ensuite dans un système AUTOSAR nécessitent des tests de non-régression et de validation pour s'assurer concrètement que la composabilité est respectée.

Si l'on souhaitait se passer de tels mécanismes en conservant les mêmes niveaux de certitudes sur les durées d'exécution des tâches, cela impliquerait un surdimensionnement déraisonnable des processeurs utilisés.

1.4.2 Mécanismes Réactifs

À l'inverse, les stratégies réactives se basent sur l'observation de l'état du système pendant son fonctionnement de façon à agir en conséquence uniquement si nécessaire.

Le principe est de monitorer en temps-réel l'exécution des processus et activer sur demande des mécanismes de prévention des fautes. Ce type de méthode est plus complexe à mettre en place et présentent a priori des garanties plus faibles. Cela en fait une solution plus adaptée pour du temps-réel souple tout en conservant des performances moyennes convenables.

Les systèmes de **watchdog** sont à la base de ce genre de mécanisme, en levant un traitement d'erreur en cas de constat d'une défaillance, de façon à isoler cette dernière. Cela ne permet pas d'empêcher l'erreur, uniquement de prévenir ou au moins mitiger toute conséquence supplémentaire.

Les techniques d'ordonnancement dynamique des tâches permettent aussi en un sens d'optimiser l'utilisation des ressources de calcul en priorisant l'exécution au plus urgent par exemple. C'est le cas par exemple d'un ordonnancement des tâches en EDF - "*Earliest Deadline First*", autrement dit "Priorité à l'Échéance au plus Tôt" qui exécute, comme son nom l'indique, systématiquement la tâche dont l'échéance est la plus proche. De façon générale, tout mécanisme de changement dynamique de priorité selon l'état du système entre dans cette catégorie.

Enfin, les mécanismes de réaction consistent essentiellement à suspendre toute tâche indésirable de façon à isoler les tâches temps-réel en cours d'exécution. Par conséquent, cela prévient pendant l'exécution les risques d'interférences matérielles, au détriment temporaire des tâches non critiques. Cela n'est bien entendu possible que dans un cadre où l'on peut modifier en fonctionnement l'exécution des tâches et se permettre d'en stopper une partie. Certains mécanismes réactifs sont à usage unique dans le sens où une fois ce déclenchement fait, le système reste dans un fonctionnement en mode dégradé pour tout le reste de son exécution. À l'inverse d'autres solutions proposent un mode dégradé temporaire avec un retour en fonctionnement nominal une fois le risque passé.

La plus grande difficulté de ces stratégies réside donc dans la preuve des garanties qu'elles sont capables de fournir sur les propriétés de sûreté de fonctionnement au regard des exigences non fonctionnelles définies. Elles permettent de mieux exploiter les ressources disponibles. Les systèmes réactifs sont par exemple à la base des usages informatiques grand public qui reposent sur un système d'exploitation standard pour lesquels il n'y a pas de contrainte temporelle dure.

1.5 Contribution de la thèse et objectifs

Dans le cadre de l'utilisation de calculateurs multicoeurs dans les applications industrielles, nous aborderont dans cette thèse les difficultés que cela implique en terme d'implémentation pour continuer à garantir le bon fonctionnement du logiciel. Plus spécifiquement, nous nous intéresseront aux implications du partage de ressources sur les temps d'exécution de logiciel critiques dans le cadre de systèmes à criticité mixte. Ces partages pouvant entraîner des congestions qui en conséquence ajoutent des latences qui peuvent aller jusqu'à provoquer des dépassements d'échéance temporelle et donc des fautes logicielles temporelles transitoires. Pour empêcher ce risque potentiellement critique, il sera proposé un mécanisme novateur de Surveillance et de Contrôle d'exécution logiciel. Les objectifs ici sont multiples, car d'une part,

l'on souhaite conserver des garanties sur les temps d'exécution de logiciel critique, mais en même temps il faut trouver des mécanismes les moins intrusifs possibles sur l'exécution pour profiter au maximum des puissances de calcul multicoeur mis à disposition.

Pour cela, nous verrons dans le chapitre 2 sur l'État de l'Art les différents propositions existantes qui permettent de répondre à tout ou partie des objectifs susmentionnés. Nous verrons cela avec un point de vue relativement général sur les différents sous-domaines de recherche afférents aux enjeux des systèmes à criticité mixte. L'objectif est d'identifier le positionnement de nos travaux au sein d'un domaine aussi vaste qui touche à de très nombreux éléments des architectures logicielles et matérielles.

Par la suite le chapitre 3 présentera notre façon d'aborder la question avec nos hypothèses et modélisation du système. Dans le cadre d'un système multicoeur qui héberge des applications à criticité mixte, on verra le modèle d'exécution adopté, orienté vers une approche originale basé sur des chaînes de tâches. Cela va impliquer des notions de précédence d'exécution et de temps de réponse bout-à-bout qui seront essentiels par la suite.

Cela nous mènera dans le chapitre 4 suivant à développer notre mécanisme de Surveillance et Contrôle basé sur ces chaînes, son architecture et ce que cela implique en terme d'implémentation. Cette approche se basera sur la surveillance des contraintes temporelles de chaînes de tâches dans un système à criticité duale, exécuté sur un multicoeur bien entendu. L'objectif étant de stopper temporairement des tâches non critiques pour éviter des interférences qui risqueraient de provoquer des fautes temporelles.

Enfin, nous verrons au chapitre 5 un cas d'implémentation qui a pu être réalisé sur une plateforme d'essai. Cette plateforme se voulant être une preuve de concept, l'objectif est de voir l'influence et les tenants et aboutissants du mécanisme proposé. Nous utiliserons pour cela des tâches d'une suite de benchmark sur laquelle on pourra implémenter le mécanisme, le calibrer et en tirer des mesures de performance. Nous ferons alors un bilan des résultats obtenus avec des perspectives d'utilisation, ainsi que des pistes de recherche.

CHAPITRE 2

État de l'Art

Sommaire

2.1	Vue d'ensemble des systèmes à criticité mixte	28
2.1.1	Entre garanties et optimisation	28
2.1.2	Solutions Matérielles et Logicielles	30
2.1.3	Contrôle Statique et Dynamique	31
2.2	Mécanismes de contrôle réactif	34
2.2.1	Contrôle réactif spatial	34
2.2.2	Contrôle réactif temporel	35
2.3	Positionnement des travaux	37

Dans le chapitre précédent, nous avons pu identifier en détails les problématiques émergentes provoquées par l'utilisation de calculateurs multicœurs dans une optique d'agrégation de logiciels à criticités mixtes. Cet ensemble de logiciels aux usages divers, et donc aux exigences de sûreté de fonctionnement variées mène à de nombreux choix et compromis qui recouvrent beaucoup d'éléments spécifiques du système. Cela débute dès le choix de l'architecture matérielle dont les processeurs à utiliser et va jusqu'au choix de l'ordonnancement des tâches et la gestion des périphériques. Tous ces éléments vont jouer sur le bon fonctionnement de l'ensemble, et notamment sur la bonne exécution du logiciel pour réaliser ses fonctions.

Dans ce contexte, plusieurs enjeux se heurtent les uns aux autres. Nous avons d'une part le besoin grandissant à l'origine de cet usage des calculateurs multicœurs : exploiter au maximum de la puissance de calcul pour réduire le nombre de calculateurs et donc les coûts. Mais d'autre part, la cohabitation de logiciels à criticités multiples requiert des garanties de sûreté de fonctionnement dont des garanties de non-interférences entre les logiciels pour s'assurer notamment de l'absence de défaillances temporelles par dépassement d'échéances.

De par la multiplicité des choix et donc des solutions envisageables au développement et l'implémentation d'une part ; de par la multiplicité des objectifs à respecter dans la réalisation des systèmes d'autre part ; il existe une infinité de combinaisons possibles pour mettre en place un système à criticité mixte sur une architecture multicœur.

Nous présentons dans ce chapitre une vue d'ensemble des différents études académiques propres aux problématiques des systèmes à criticité mixte en les classifiant par grands axes méthodologiques. Cela nous permettra d'avoir une vue d'ensemble de ce sujet de recherche qui est si vaste. À partir de cela, nous focalisons le point sur lequel se concentre nos travaux et comment nous nous positionnons dans ce champ des possibles.

2.1 Vue d'ensemble des systèmes à criticité mixte

2.1.1 Entre garanties et optimisation

La maîtrise des processeurs multicœurs est un domaine de recherche très vaste qui se subdivise assez rapidement en deux objectifs principaux comme nous avons déjà pu le présenter. D'un côté, nous avons la recherche du maximum de performance à partir des ressources de calcul à disposition et notamment en profitant du parallélisme. De l'autre côté, nous avons l'analyse de la dimension temporelle pour la garantie de respect des exigences temporelles. Il s'avère que ces deux objectifs peuvent être antagonistes.

Historiquement, cette division s'est naturellement répartie selon le domaine d'utilisation de l'électronique embarquée et des contraintes associées. L'optimisation des ressources étant en premier lieu associé avec des usages grand public tel que sur nos ordinateurs et autres smartphones, que l'on peut se permettre de redémarrer en cas de problèmes, et qui se doivent de s'adapter du mieux possible à la charge variable qu'on leur impose. Plus généralement, une très grande part des recherches pour le développement des systèmes d'exploitation comme Linux ou Android sont orientées vers un meilleur usage du hardware. Cela passe en premier lieu par l'ordonnancement avec le *completely fair scheduling* [Pabla 2009], [Pricopi 2014] par exemple, mais aussi par l'allocation des tâches et l'équilibrage de charge des cœurs [Pathania 2016]. Ce besoin devient de plus en plus fort avec l'utilisation de services cloud décentralisés [Walsh 2004]. Il s'agit là d'un problème ancien et en constante évolution face à l'incessante croissance en puissance et en complexité des processeurs [Lozi 2016]. De plus, à cette problématique s'ajoute des sous-enjeux qui deviennent de plus en plus importants tels que la gestion de la chauffe par répartition matérielle de la charge ou encore la gestion de la consommation énergétique [Li 2016]. De par ses caractéristiques de systèmes hautement évolutifs, dynamiques, au contact constant avec l'utilisateur et donc en constante évolution, j'aime à qualifier ces systèmes de *systèmes flexibles*. Il s'agit d'avoir la plus grande adaptabilité possible pour satisfaire à l'utilisateur, en garantissant au mieux une expérience utilisateur et une disponibilité satisfaisante, mais avec des exigences de fiabilité qui sont rarement le critère central de décision dans le processus de développement.

De l'autre côté du spectre, nous avons les cas d'application industriels qui ont des besoins de fiabilité et de sûreté de fonctionnement à l'utilisation bien plus exigeants. C'est dans cette branche-là que se focalisent sans nul doute la plus grande part des recherches de par les enjeux financiers et technologiques qui sont impliqués. C'est d'autant plus vrai que les enjeux industriels rencontrent de plus en plus les besoins qui étaient jusqu'alors propres aux systèmes flexibles. Les besoins en performance s'accentuent, ainsi que les besoins en évolutivité de par la cohabitation grandissante entre les systèmes en contact avec l'utilisateur et les systèmes enfouis. L'automobile, l'avion ainsi que tous les systèmes embarqués qui s'interfacent de plus en plus avec des smartphones en sont des exemples flagrants. À l'aune de ces faits, ces systèmes ont pour caractéristique émergente d'héberger du logiciel à criticité mixte, avec d'une part des composants originairement rencontrés dans les systèmes flexibles et d'autre part les composants plus stables et sûrs de fonctionnement rencontrés dans

les systèmes embarqués industriels. Dans ce contexte, des revues ont d'ores-et-déjà été réalisées sur le champ des propositions du monde de la recherche pour gérer des systèmes à criticité mixte. Il nous faut citer l'incontournable Review de Burns et Davis [Burns 2022] qui présente de façon synthétique les différentes solutions à cette problématique.

L'objectif de cette étape est de nous situer dans cet univers tentaculaire des systèmes à criticité mixte. Nous proposons donc une grille de lecture par branches macroscopiques de recherche telle que décrite dans la Figure 2.1, dont nous venons de présenter le premier étage qui distingue deux objectifs fondamentaux qui sont les garanties temporelles et l'optimisation des performances. Ces deux principes à première vue opposés se doivent pourtant de plus en plus d'être conciliés comme on a pu le présenter au cours de notre constat au chapitre d'introduction. À cette fin, nous décidons d'orienter notre réflexion en premier lieu sur ce qu'il est possible de garantir sur les exigences temporelles. Il s'agit d'aborder la problématique sous un angle qui permettrait un large panoramique d'opportunités d'améliorations que ce soit pour y adjoindre des mécanismes de maximisation d'usage du processeur ou à l'inverse des outils complémentaires pour la maîtrise d'exécution des logiciels critiques. De cette façon, on serait en mesure de proposer un cadre qui apporte un certain nombre de garanties minimales, auquel il est possible d'adjoindre d'autres solutions complémentaires, notamment parmi celles que l'on présente ici.

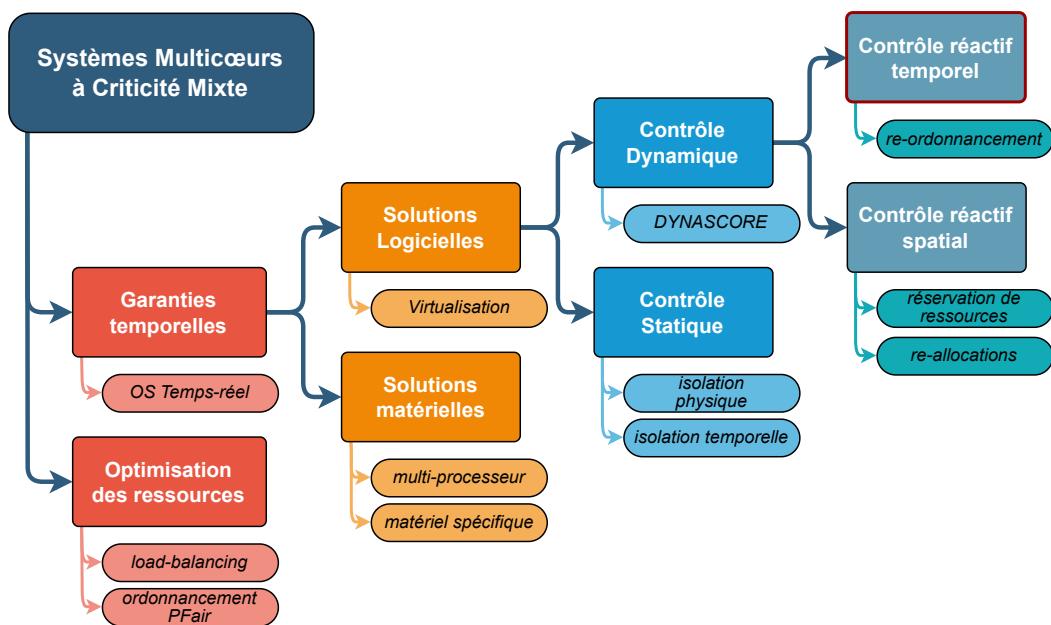


FIGURE 2.1 – Vue d'ensemble des solutions pour systèmes à criticité mixte sur calculateur multicœur

2.1.2 Solutions Matérielles et Logicielles

2.1.2.1 Solutions Matérielles

Parmi les axes existants pour maîtriser l'exécution de systèmes à criticité mixte, le premier choix repose dans la mise en place de mécanisme d'inhibition du problème initial. Il s'agit de gérer le problème à la source en prévenant purement et simplement toute possibilité d'interférence logicielle. Il s'agit probablement encore aujourd'hui de l'option la plus répandue dans l'industrie pour sa simplicité. Cela est dû à la simplicité du résultat obtenu notamment quand il s'agit de certifier le logiciel. En étant capable de prouver que 2 logiciels de niveau de criticité différents n'entreront jamais en concurrence pour l'utilisation du matériel, les critères de sûreté de fonctionnement pour la certification sont directement atteints.

Il est possible de garantir l'absence totale d'interférences soit de façon logicielle par des mécanismes de contrôle de l'exécution, soit de façon matérielle, par construction. Ainsi, le choix fondamental de l'architecture matérielle peut directement répondre au problème. De fait si l'on est en capacité d'exécuter directement le logiciel sur un support qui, par construction, ne présente aucun risque d'interférences lié au partage de ressources, alors le problème est directement évité. C'est ce que tentent de proposer des travaux comme [Schoeberl 2011]. L'autre possibilité repose tout simplement sur ce qui est réalisé aujourd'hui dans l'automobile par exemple, en répartissant le logiciel dans plusieurs processeurs, en utilisant donc des méthodes de communication et synchronisation inter-processeur dans une architecture distribuée.

C'est aussi l'approche abordée avec les calculateurs scratchpad que nous avons présenté précédemment (c.f. chapitre d'Introduction), basé sur une mémoire dédiée à chaque cœur et des séparations fortes pour prévenir toute contention. L'inconvénient fondamental de ces solutions réside dans leur immuabilité. Les perspectives d'évolution sont restreintes par le matériel, à moins d'être complètement remplacé. Par ailleurs, la conception et mise sur le marché de ce type de matériel spécialisé dépend du bon vouloir des fondeurs et pose en conséquence aussi des problèmes de coûts. Il existe par ailleurs des solutions matérielles plus fines comme avec l'ajout de modules hardware dédiés qui sont directement connectés au processeur pour accomplir des tâches spécifiques [Solet 2018].

Une analyse très complète du sous-domaine des solutions matérielles peut être consultée dans la thèse de A. Blin [Blin 2017].

2.1.2.2 Solutions Logicielles

Inversement, il est possible de répondre aux problèmes d'interférences par des solutions logicielles. C'est dans ce cadre-ci qu'une très grande part des recherches sont effectuées. De fait un grand nombre de paramètres logiciels permettent d'influer sur les performances d'exécution de tâches concourantes. On pourra citer notamment les stratégies pour :

- l'ordonnancement des tâches,
- l'allocation des tâches sur les coeurs,
- l'accès aux ressources partagées (caches, mémoire, bus d'accès...).

Tous ces éléments ayant des interactions fortes entre eux, les solutions logicielles proposent à la fois une très grande variété de choix et de possibilités d'aborder la question, mais cela pose par la même la question de la complexité de réalisation qui dépendra du niveau de focalisation sur un point précis ou à l'inverse de la manipulation de tous les aspects du système. Les solutions les plus globales se retrouvent notamment dans les solutions de virtualisation [Augier 2006] et plus généralement des systèmes d'exploitation dédiés temps-réel (RTOS – Real-Time OS) comme PikeOS [Kaiser 2007] ou encore OSEK/VDX [Bechenne 2006] qui est un standard ayant donné lieu à plusieurs implémentations. Le cadre offert par ces propositions ne répond a priori pas directement aux problématiques émergentes d'interférences entre les logiciels. En revanche ils proposent tous les outils d'implémentation nécessaires. Ce sont donc des technologies probablement nécessaires, mais pas suffisantes, à l'implémentation et certification de systèmes temps-réels à criticité mixte qui requièrent la mise en place de mécanismes plus spécifiques. On pourra mentionner le cas de PikeOS qui a proposé récemment la mise en place d'une structure de la plateforme opérationnelle avec allocation de ressources temporelles (plages de temps fixes) et spatiale (plages d'utilisation de chaque ressource partagée) pour l'exécution des tâches critiques et non critiques de façon à prévenir toute exécution simultanée qui pourrait engendrer des interférences entre ces deux catégories de logiciels [Sysgo AG 2019]. Cela a permis en 2013 la certification au plus haut niveau pour le ferroviaire (SIL4 de la norme EN50128) du framework temps-réel ainsi développé pour un processeur dual cœur.

C'est dans ce cadre que bon nombre d'études analytiques du problème sont développées. En effet, un des ressorts principaux de la maîtrise de l'exécution du logiciel réside dans le maintien d'un mode de fonctionnement nominal associé à des estimations de pire temps d'exécution (*WCET – Worst Case Execution Time*). Il s'agit des fondamentaux d'analyse des systèmes à criticité mixte, tel que présenté par [Vestal 2007]. Chaque niveau de criticité pouvant être associé avec différents degrés d'exigence sur l'estimation des WCET. Sachant que plus une estimation de WCET est précise et exhaustive dans ce qu'elle prend en compte, plus ce sera complexe et coûteux à réaliser. De fait, l'estimation précise et sûre du WCET d'une tâche s'exécutant sur un processeur multicœur reste aujourd'hui un problème ouvert, encore abordé dans le cadre d'outils d'estimation analytique comme [Kästner 2019]. De ces estimations peuvent découler diverses stratégies de contrôle de l'exécution du logiciel.

2.1.3 Contrôle Statique et Dynamique

Il est possible d'établir une dichotomie parmi les mécanismes de gestion des tâches dans les systèmes à criticité mixte entre les solutions *statiques* d'une part et les solutions de contrôle *dynamique* d'autre part.

2.1.3.1 Contrôle statique

Dans le cas d'utilisation de solutions statiques, les décisions d'exécution des tâches sont prévues en amont, lors de la phase de développement, pour obtenir un

modèle totalement prédictif et stable d'exécution. Ils se basent donc sur la combinaison des spécifications fonctionnelles et sur des estimations de temps d'exécution moyen et pire temps pour prédéterminer un ordonnancement et une allocation statique des ressources. Cela permet d'éviter par construction l'exécution concourante de logiciels qui pourraient partager des ressources

Il s'agit de ce qui est le plus communément employé, avec des surestimations nécessaire des créneaux réservés à chaque tâche de façon à garantir en toute condition le respect des exigences temporelles en évitant toute exécution parallèle indésirable. Dans l'industrie les frameworks employés se basent sur ce type de méthodes. Il y a notamment Classic AUTOSAR [AUTOSAR 2016] dans l'automobile qui utilise un ordonnancement fixe, ou encore dans l'avionique avec le standard ARINC 653 [Prisaznuk 2008]. Toujours dans l'avionique, le réseau de communication standardisé AFDX [Charara 2006] utilise aussi une méthode d'allocation statique de créneaux de communication pour obtenir des garanties fortes sur la latence d'émission/réception des messages. En contrepartie de la forte maîtrise de l'exécution du logiciel, ces méthodes ont l'inconvénient de sur-allouer les ressources nécessaires au fonctionnement du système, et donc d'aller à l'encontre de l'objectif de maximisation d'utilisation de ces dernières.

En plus des stratégies d'ordonnancement statiques qui présentent une forme d'isolation temporelle entre les tâches, il est possible de proposer de l'isolation spatiale, par le partitionnement des ressources mémoire notamment. On peut citer par exemple le travail de [Mancuso 2013] qui propose une méthode de profilage hors-ligne des tâches pour déterminer les pages mémoires les plus utilisées . Cela permet de contrôler la position de ces données dans les caches partagés pour limiter les interférences associées. D'autres travaux se sont intéressés plus spécifiquement à la question de l'allocation des tâches en complément d'une isolation spatiale et temporelle pour optimiser la taille des fenêtres temporelles réservées pour chaque tâche [Tamas-Selicean 2011]. Enfin, certains travaux ont combiné l'utilisation de méthodes d'ordonnancement statique des tâches par exclusion d'exécution simultanée des tâches de différents niveaux de criticité avec des techniques d'isolation matérielle suivant des méthodes analytiques d'identification des interférences pour proposer une approche générale qui a pu être mise en application sur un cas avionique réel. Cette approche hybride mi-statique mi-adaptative semble présenter des résultats encourageants [Giannopoulou 2013].

L'association de partitionnement temporel et spatial des tâches avec des méthodes comme celles susmentionnées permet d'obtenir les exigences nécessaires pour la sûreté des systèmes embarqués tels que décrit dans les divers standards associés. Ils permettent en revanche peu d'évolutions sans avoir à refaire tout le travail d'intégration depuis le départ.

De fait, il a été admis qu'il semble difficile, sinon impossible, de trouver une méthode qui capture les problèmes de contention dans leur ensemble dans un système multicœur avec cache partagé [Suhendra 2008]. Cela mène les solutions suscitées à devoir surestimer les réservations de ressources nécessaires à l'obtention des garantis temps-réels. Ce constat nous conforte dans l'idée qu'à défaut de pouvoir contrôler et empêcher tout risque de contention dans ce type d'architecture, il est préférable d'en surveiller et contrôler les conséquences pour prévenir toute conclusion

catastrophique.

Il est à noter que cela ne disqualifie pas pour autant ces méthodes, qui apportent des propriétés solides. Une solution globale consistera sans aucun doute en un juste milieu entre l'usage de mécanismes statiques d'allocation et d'isolation statique des logiciels avec des mécanismes réactifs d'adaptation pour permettre malgré tout une meilleure utilisation des ressources de calcul.

2.1.3.2 Contrôle Dynamique

Survient alors la possibilité d'employer des mécanismes de monitoring pendant l'exécution de façon à pouvoir réagir à des évolutions du système qui mènent à des défaillances temporelles tel qu'un dépassement d'échéance. L'intérêt de ce type de mécanisme est de privilégier l'adaptation aux conditions de fonctionnement. Cela peut permettre une meilleure robustesse à des légères variations dans les conditions d'usage. Ces aléas de fonctionnement peuvent être de nombreuses origines, à la fois externe avec des conditions de fonctionnement qui n'ont pas été prévues (interférences électro-magnétiques, comportement utilisateur inattendu...) ou bien interne (conjonction de plusieurs événements logiciels simultanés qui donnent un comportement imprévu). Les mécanismes réactifs sont en revanche plus complexes à concevoir pour continuer à offrir les garanties désirées et apportent forcément leur lot d'incertitudes.

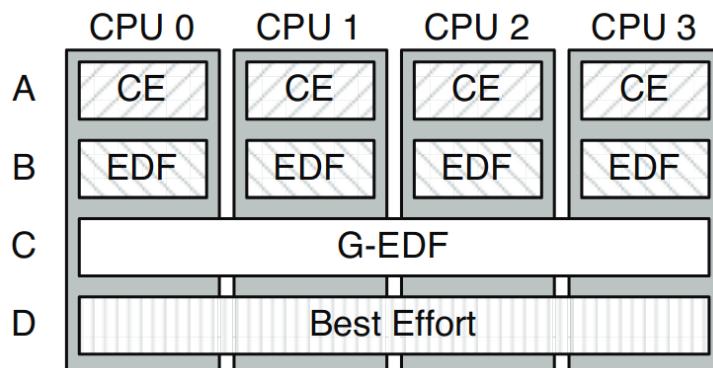


FIGURE 2.2 – Allocation de conteneurs avec leurs politiques d'ordonnancement pour chaque niveau de criticité avec MC² sur un quadricœur [Herman 2012]

Une des premières approches pour l'implémentation de systèmes à criticité mixte est le framework MC² [Anderson 2009] dont une implémentation est proposée par [Herman 2012]. Ce dernier propose un cadre d'implémentation pour un système à 5 niveaux de criticité¹ qui disposent chacun d'un conteneur dédié ainsi que d'une politique d'ordonnancement associée tel qu'illustré dans la Figure 2.2. Le concept sous-jacent est de prioriser les niveaux de criticité par étage. Ainsi, la partition de plus haut niveau de criticité est exécutée en priorité, avec un ordonnancement statique selon une table préétablie (*cyclic executive*). Quand aucune tâche de ce niveau de criticité n'est en attente d'exécution, les tâches du second niveau de

1. les niveaux de criticité 3 et 4 décrits ont été regroupés dans le schéma de Herman & al.

criticité peuvent s'exécuter selon un ordonnancement pEDF (*partitionned Earliest Deadline First*), c'est-à-dire priorité à l'échéance la plus proche, partitionné selon chaque cœur. De la même façon, les troisième et quatrième niveaux s'exécutent si aucune tâche des niveaux de criticité supérieurs ne sont en attente, cette fois-ci selon une politique g-EDF (*global-EDF*). Et uniquement si tous ces niveaux de criticité se sont exécutés sans dépassement, alors le dernier niveau de criticité peut s'exécuter en *best-effort*.

D'autres travaux ont pu montrer que du contrôle dynamique par construction de modèles prédictifs d'exécution présente aussi une bonne piste d'optimisation de l'exécution des applications sur multicœur en limitant les interférences. Des travaux comme ceux de [Kim 2019] ont pu étudier la question par l'utilisation d'un modèle par apprentissage automatique. C'est par conséquent vers ce type de solutions réactives avec un contrôle dynamique de l'exécution de tâches que nous allons nous positionner.

2.2 Mécanismes de contrôle réactif

Les mécanismes de contrôle réactif peuvent se focaliser sur différentes parties du système pour limiter les risques de fautes logicielles provoquées par l'exécution de logiciels concurrents. D'ailleurs rien n'empêcherait d'employer des mécanismes qui agiraient sur plusieurs composants à la fois. Les principaux éléments sur lesquels les mécanismes de contrôle agissent sont :

- directement sur l'ordonnancement des tâches, par mise en pause ou changement de priorités des tâches durant le fonctionnement. Il s'agit alors d'un contrôle dynamique *temporel* (c.-à-d. le *moment* où s'exécutent les tâches est ajusté par rapport au fonctionnement nominal).
- sur l'utilisation du support d'exécution soit par limitation d'accès (budget d'utilisation), soit par priorisation de l'utilisation, soit par réservation d'espace dédié à certaines tâches pour les espaces mémoires notamment. Il s'agit alors d'un contrôle dynamique spatial (c.-à-d. la capacité d'accès au support d'exécution, et donc aux ressources partagées, est ajusté par rapport au fonctionnement nominal).

2.2.1 Contrôle réactif spatial

Le contrôle réactif spatial repose sur la gestion des ressources partagées de façon à isoler les tâches de différents niveaux de criticité sur l'utilisation de ces dernières. Il peut s'agir d'allocation d'espace mémoire avec réservation de cache par exemple [Suhendra 2008]. D'autres travaux se focalisent plutôt sur les politiques d'accès aux ressources via les bus d'accès.

En ce sens, des travaux comme ceux de [Blin 2016a] proposent un mécanisme de monitoring qui a été calibré hors-ligne, en amont, de façon à identifier une surcharge du système. Le cas échéant, un contrôle spatial est effectué pour rendre l'utilisation du bus d'accès mémoire exclusif au cœur exécutant les tâches critiques pour garantir le respect des échéances sur ces dernières. La solution ici proposée est intéressante

dans son approche, mais de par ses hypothèses implique une mise en pause de coeurs entiers du processeur, qui auront été alloués aux tâches non critiques. De même, [Yun 2012] propose une régulation de la bande passante au niveau système pour chaque cœur.

Il existe par ailleurs des mécanismes réactifs qui vont modifier l'allocation dynamique des tâches sur les coeurs pour équilibrer la charge. C'est le cas du modèle proposé dans [Xu 2019] qui suppose un changement de mode d'une partie des coeurs, permettant la migration de tâches non critique vers les coeurs qui sont toujours en fonctionnement nominal.

2.2.2 Contrôle réactif temporel

Nombreux sont les travaux qui réalisent du contrôle dynamique temporel, axé sur une adaptation de l'ordonnancement des tâches. Les ordonnancements dynamiques tels que les dérivés d'EDF [Lelli 2011], [Behera 2012], [Rodriguez 2013] permettent l'optimisation des ressources de calcul. La question demeure sur l'usage de stratégies d'ordonnancement globaux (l'exécution des tâches et leur allocation aux coeurs disponibles se fait à l'exécution) ou bien partitionnés (chaque cœur dispose de son propre ordonnanceur, les tâches sont allouées à chaque cœur à la conception). Ces deux grandes méthodes d'ordonnancement sont représentées dans la Figure 2.3. Il existe par ailleurs des stratégies d'ordonnancement intermédiaires dites semi-partitionnées où certaines tâches sont partitionnées tandis que d'autres peuvent migrer librement entre les coeurs. Des comparaisons ont pu être étudiées entre ces types d'ordonnancement, par exemple par [Li 2014].

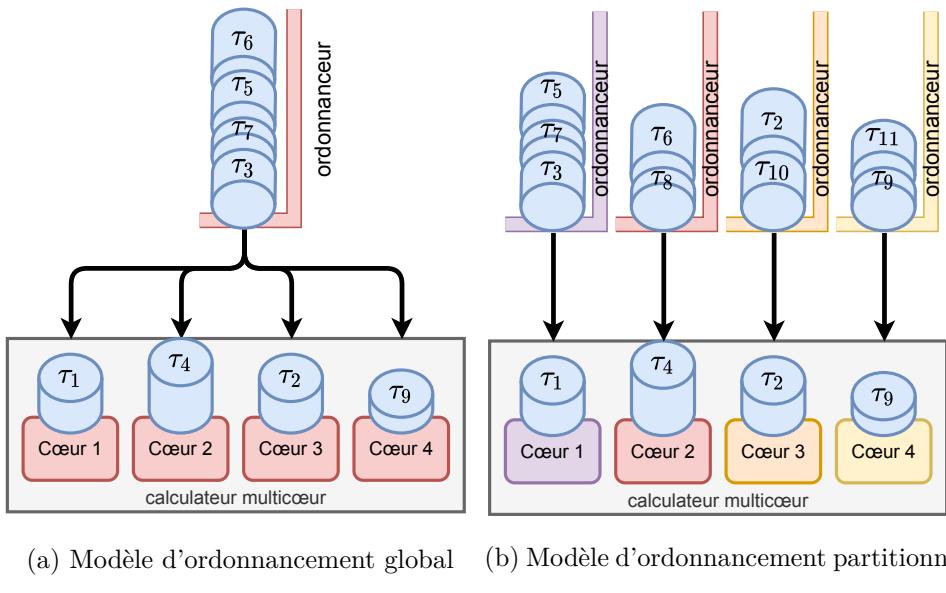


FIGURE 2.3 – Modèles d'ordonnancement

Aussi, il est possible de disposer d'un ordonnancement des tâches différent pour plusieurs modes de fonctionnement du système. Ce dernier change donc de mode selon les besoins ou les risques. Un modèle spécifique de système à criticité mixte

ainsi proposé est le modèle à *criticité duale*. En d'autres termes, les tâches sont catégorisées selon deux niveaux de criticité différents. Selon les cas, on peut parler de criticité haute et basse, de tâches strictes et souples ou encore simplement de tâches critiques et non critiques. Les solutions concernées dans ces situations reposent alors sur un passage d'un mode de fonctionnement faiblement critique vers un mode de fonctionnement hautement critique où seul l'exécution des tâches de criticité haute est garantie. C'est le cas par exemple de la proposition d'amélioration de MC² avec un changement de mode dans [Chisholm 2017]. Dans une vision axée sur la sûreté de fonctionnement des systèmes, on peut considérer que le mode de fonctionnement faiblement critique est le comportement nominal attendu. À l'inverse, les modes plus critiques sont des modes dégradés progressifs dans lesquels certaines tâches peuvent avoir un fonctionnement non garanti, voire abandonné. Ces modes de fonctionnement ne doivent en conséquence pas constituer la règle, mais l'exception, envisagée pour les exigences de sûreté de fonctionnement. Les travaux de Trapp et al. ont eu l'occasion de s'intéresser à la pertinence d'utiliser de tels mécanismes de changement de modes [Trapp 2007]. Il s'agit d'un élément relativement important à mentionner dans le cadre d'une proposition de mécanisme qui apporte une garantie stricte dans des situations pire cas, en complément d'une infrastructure logicielle qui propose un cadre d'exécution de logiciel à criticité mixte avec une bonne exploitation des ressources.

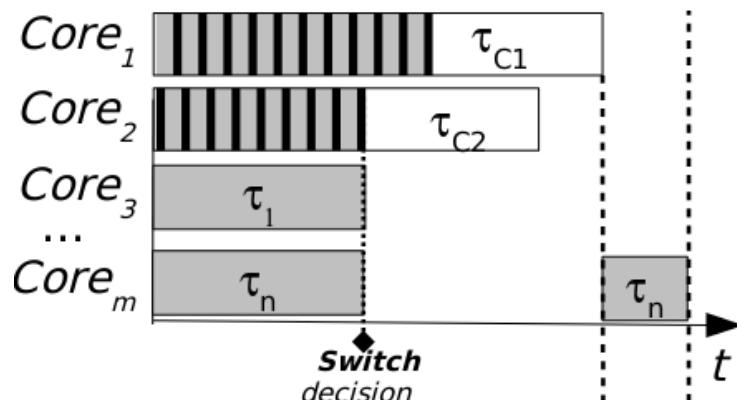


FIGURE 2.4 – Principe de contrôle d'exécution non optimisé dans [Kritikakou 2016]

Dans ce sens, [Kritikakou 2017] propose un mécanisme réactif qui se base sur deux niveaux de criticité avec le diagnostic de risque d'interférences des tâches à faible niveau de criticité sur les tâches critiques du fait du matériel partagé. La proposition consiste à surveiller l'exécution des tâches critiques de façon à identifier le moment où les interférences ne peuvent plus être tolérées, au risque de déclencher des dépassements d'échéances critiques. À cet instant-là, les tâches non-critiques sont stoppées pour prévenir le risque. La méthode de contrôle consiste à mesurer à des points de fonctionnement prédéfinis l'état d'avancement de l'exécution des tâches critiques, et de le comparer à des estimations de pire temps d'exécution restant à ce point de fonctionnement. Par calcul d'une condition de sûreté, le mécanisme de contrôle identifie le risque pour passer en mode dégradé. Tel qu'illustré en Figure 2.4,

les tâches critiques τ_{C1} , τ_{C2} sont exécutées sur les coeurs 1 et 2 tandis que les tâches non critiques τ_1 [...] τ_n sont exécutées sur les autres coeurs. La vérification à intervalle régulière permet d'identifier le point de décision au-delà duquel les tâches non critiques sont arrêtées temporairement pour garantir la terminaison des tâches critiques avant leur date butoir.

Ces résultats ont été une grande source d'inspiration pour la suite de ces travaux dans le concept de surveillance de l'exécution des tâches critiques pour ne provoquer une transition dans un mode dégradé qu'en cas de dernier recours. Cela offre l'avantage d'être accessible quels que soient les autres mécanismes et méthodes employées par le système pour l'exécution des tâches, tout en offrant des garanties claires pour certaines d'entre elles. En revanche, on pourrait reprocher le fait que cette méthode requiert l'instrumentation des tâches au niveau de leur code source, ce qui n'est pas possible dans le cadre d'emploi de logiciels en boîte noire.

2.3 Positionnement des travaux

Dans la vaste étendue des études sur les systèmes à criticité mixte et de sûreté de fonctionnement, nous avons proposé une classification macroscopique. Bien entendu, cette dichotomie itérative est arbitraire et certains types de mécanismes n'y sont par conséquent pas positionnables. Par exemple, il est possible de trouver des mécanismes matériels dynamiques qui se basent sur une reconfiguration matérielle en cas de faute temporelle, tel que dans [Lin 2015] avec la prise en compte d'un calculateur primaire et secondaire pour le changement de mode. Ceci étant dit, la classification que nous proposons, agrégée dans la Figure 2.1 permet d'avoir une approche du domaine du plus global au plus spécifique par segmentation des propositions académiques en dichotomies successives. Il nous est ainsi possible de positionner notre axe d'approche dans cet ensemble et d'identifier les choix techniques et méthodologiques associés à chaque embranchement de cette vue d'ensemble.

En effet, au regard des éléments précédemment mentionnés, il nous semble intéressant de trouver une solution logicielle qui soit *flexible* et *combinable* facilement avec d'autres techniques existantes, qui abordent potentiellement le problème sur un autre de ses paramètres. Plus spécifiquement, il devrait être possible de proposer des méthodes d'ordonnancement et d'allocation des tâches qui permette une bonne utilisation des ressources matérielles, tout en y adjoignant notre proposition qui offre des *garanties sur les échéances temporelles*. Ajouté à cela la complexité d'identification et neutralisation des interférences matérielles, cela impose un *mécanisme logiciel réactif* d'anticipation et mitigation des fautes qui influe sur l'exécution des tâches non critiques. Aussi, il devient souhaitable de permettre un retour en fonctionnement nominal, dans les conditions où les risques de défaillances ont été écartés.

Par ailleurs, nous sommes dans un contexte industriel où les évolutions logicielles sont constantes, avec des contraintes de coûts et de temps de développement non négligeables. Ce contexte implique aussi l'usage de logiciels en boîte noire, où il n'est pas possible d'accéder et modifier le code source de façon systématique. Cela demande donc des solutions qui demandent un surcoût de développement limité en

cas d'évolution (typiquement associé à le re-estimation analytique précise de pires temps d'exécution par exemple) et employable sans modification de code source.

Avec la considération de tous ces éléments, notre objectif est :

- d'éviter les défaillantes temporelles par dépassement d'échéances critiques d'une part,
- de permettre une bonne utilisation des ressources matérielles d'autre part.

Par conséquent, nous proposons un mécanisme de surveillance et de contrôle de l'exécution des tâches critiques, de façon à intervenir sur l'exécution des tâches non critiques uniquement en cas de nécessité. L'intention étant de prévenir de façon préventive et circonstancielle les interférences inter-tâches pour offrir des garanties temporelles aux tâches critiques. L'approche se voudra non intrusive sur le logiciel embarqué en abordant les tâches critiques d'un point de vue fonctionnel avec un mécanisme de contrôle bas niveau pour la surveillance et le contrôle de l'exécution.

CHAPITRE 3

Principe et architecture

Sommaire

3.1 Modèle basé sur des chaînes de tâches	40
3.1.1 Notion de Criticité	41
3.1.2 Modèle de Tâche et Chaînes de tâches	43
3.2 Mécanisme d'anticipation	49
3.2.1 Méthode d'anticipation	49
3.2.2 Passage en Mode Dégradé	51
3.3 Architecture Logicielle	53
3.3.1 Task Wrapper Component (TWC)	55
3.3.2 Core Control Component (CCC)	56
3.3.3 Définition des constantes de Contrôle	57
3.4 Application industrielle	57
3.4.1 Spécification fonctionnelle	57

De fait et pour rappel, l'objectif principal de ces travaux est de proposer un mécanisme logiciel qui permette dans le même temps à utiliser au maximum les ressources matérielles disponibles et avoir des garanties minimales sur les temps d'exécutions. Tout l'enjeu de cette proposition est d'arriver à une solution qui limite les coûts de développement notamment en étant compatible avec des composants logiciels *legacy* et/ou en boîte noire qui ne permettent pas d'être modifié dans leur fonctionnement interne. Mais aussi en limitant les besoins de modification de l'architecture logicielle suite à des mises à jour ou l'ajout de fonctionnalités supplémentaires. Par ailleurs, on tentera de s'abstraire le plus possible d'un contexte industriel donné pour proposer une approche généraliste qui puisse s'adapter à différents domaines et selon le contexte. Dans ce chapitre, nous allons par conséquent détailler notre l'angle d'approche, avec un contrôle du logiciel à l'échelle de chaînes de tâches. L'idée est de permettre une plus grande flexibilité et permettre plus de marge de manœuvre au bon fonctionnement du système à l'échelle de contraintes temporelles bout-en-bout pour exploiter au maximum les ressources. De cette façon, on espère proposer un Mécanisme de Surveillance et Contrôle plus permissif et adapté aux contraintes préexistantes.

Nous verrons au sein de ce chapitre dans un premier temps les prérequis considérés ainsi que notre approche sur la modélisation du problème. Suite à cela nous verront dans un second temps la solution proposée qui est un mécanisme de sûreté de fonctionnement réactif de type Surveillance et Contrôle. Son objectif sera de prévenir les fautes temporelles par dépassement d'échéances sur les tâches critiques, via une approche basée sur des chaînes de tâches. Pour finir, nous verrons dans

quelle mesure notre proposition se positionne vis-à-vis des standards d'architectures existants dans différents domaines industriels dont l'automobile.

3.1 Modélisation basé sur des chaînes de tâches pour garantir les contraintes temporelles

Avec la présentation des évolutions technologiques et l'arrivée de calculateurs multicœurs complexes dans l'industrie, nous avons identifié de nouveaux enjeux d'implémentation d'architectures fédérées où des logiciels de différents niveaux de criticité doivent cohabiter. Cette cohabitation provoquant des risques supplémentaires d'interférences logicielle, peut amener à des défaillances d'exécution, notamment par dépassement d'échéances temporelles sur des tâches critiques. Pour répondre à ce problème tout en souhaitant maintenir une bonne utilisation des ressources matérielles, nous nous focalisons sur la surveillance des temps d'exécution bout-en-bout, sur une chaîne de tâche critique. Le fait d'être dans une situation à criticité mixte nous permettant de jouer sur l'exécution ou la mise en pause temporaire des tâches non critiques pour garantir le respect des échéances temporelles. Pour ce faire nous présentons dans la suite les bases pour définir un mécanisme de Surveillance-Calcul-Contrôle qui permette un passage dans un mode de fonctionnement dégradé où les tâches non critiques sont stoppées pour conserver des garanties minimales de fonctionnement, qui se caractérisent par l'exécution bout-en-bout d'une chaîne de tâche.

Afin d'étudier et développer notre mécanisme de gestion de fautes temporelles dans le cadre d'un système à criticité mixte (*Mixed Criticality Systems*), nous avons besoin de formaliser la représentation des tâches qui seront à l'étude et leur modèle. Remarquons que le modèle ici proposé se veut simple, mais réaliste pour être adapté à une réalité industrielle. De fait, on retiendra deux critères principaux pour guider le choix de notre modèle : la simplicité d'implémentation et l'accessibilité à des suites logicielles qui peuvent servir de tâches pour simuler un système réel lors de nos tests. L'objectif est ainsi de trouver un modèle qui soit à la fois suffisamment représentatif d'une réalité technique dans les milieux industriels d'une part et qui soit suffisamment légère pour obtenir une première preuve de concept fonctionnelle, maîtrisée et améliorable.

Ce modèle doit décrire d'une part la méthode d'exécution des tâches, la façon d'interagir, entre-elles, notamment pour les tâches à haut niveau de criticité qui sont reliées sous la forme d'une chaîne pour réaliser une fonction critique. Il est à noter que le mécanisme de sûreté de fonctionnement que nous proposons par la suite est *in fine* indépendant du modèle de tâche ici proposé. Il conviendra d'adapter au besoin la partie de Contrôle du mécanisme, de façon à ce qu'elle prenne en compte l'état d'exécution du système selon le modèle de tâche utilisé, s'il est différent de celui présenté ici. Typiquement la vérification des contraintes de précédence peut différer.

3.1.1 Notion de Criticité

Les systèmes à criticité mixte ont mené à de nombreuses études pour gérer la répartition temporelle des tâches, et en même temps prendre en compte les conditions de partitionnement et le partage des ressources de façon à optimiser l'usage de ces dernières. La plupart des travaux dans ce domaine se basent sur le modèle d'un ensemble de tâches τ_i caractérisées par (T_i, D_i, C_i, L_i) :

- T_i - la période d'apparition de la tâche
- D_i - la date limite d'échéance d'exécution (que l'on appelle communément *deadline*)
- C_i - Le Temps d'Exécution Pire Cas
- L_i - Le Niveau de Criticité de la tâche

Avant toute chose, il est à noter que la notion de criticité des tâches que nous avons déjà mentionnée dès le chapitre d'introduction est polysémique. En effet, selon le domaine et la spécialisation des interlocuteurs, ce mot peut traduire des enjeux de sûreté de fonctionnement différents. Il convient donc de clarifier cette notion de criticité pour notre cas avant d'aller de l'avant. Ce problème de sens a été souligné notamment par [Graydon 2013]. Il existe deux grandes approches à la notion de criticité.

La première, d'un point de vue modélisation de modèle d'exécution temps-réel, se retrouve essentiellement dans les recherches académiques. La notion de criticité est reliée aux contraintes temporelles imposées sur les tâches. Cela se traduit par le niveau de fiabilité de l'estimation des pires temps d'exécution C_i [Vestal 2007]. Plus le niveau de criticité sera élevé, plus ce WCET sera estimé de façon fiable (l'on pourrait dire de façon "prudente") et avec des contraintes fortes. Autrement dit, un haut niveau de criticité requiert de prendre en compte les cas les plus extrêmes de retard d'exécution pour estimer le WCET et les temps d'exécutions estimés pour l'ordonnancement sont plus longs. Ce type de formulation a été étendue avec des modes de criticité différente. À chaque mode de criticité est associé un niveau de criticité des tâches, plus ou moins pessimiste. Tout l'intérêt de ces modèles dans le cadre de systèmes à criticité mixte est de proposer des stratégies d'ordonnancement qui prennent en compte ces niveaux et ces modes de criticité pour permettre la priorisation d'exécution de certaines tâches plus critiques que d'autres, pour aller jusqu'à stopper des tâches moins critiques pour garantir l'exécution des tâches de criticité supérieure. Cette façon d'aborder la criticité est donc totalement orientée suivant des critères d'ordonnancement et de respect des échéances temporelles.

On notera dès cette première formulation un mélange entre les niveaux de criticité des tâches et des modes de criticité.

Le second, d'un point de vue des standards de sûreté de fonctionnement, est largement exploité dans l'industrie. Dans ce cadre-là, la criticité détermine un niveau de confiance que l'on peut accorder à un composant tel que décrit précédemment en sous-section 1.2.1. Le niveau de criticité est alors déterminé non pas au regard des contraintes temporelles, mais plutôt avec des analyses de sûreté (Analyses de Risques, FMEA, FTA...) pour obtenir une catégorisation tel que défini par les

différentes normes du domaine (un niveau d'ASIL tel que défini dans ISO26262 pour l'automobile par exemple). Elle se définit alors en fonction de critères comme :

- a) l'évaluation des conséquences en cas de défaillance,
- b) la probabilité de défaillance,
- c) les moyens disponibles pour compenser ou gérer la faute si elle survient.

Par conséquent, le niveau de criticité d'une application ne reflète alors pas nécessairement la sévérité ou les répercussions d'une faute. De fait, si l'on regarde la définition de criticité sous forme de niveaux d'ASIL dans l'automobile, tel qu'indiqué en Tableau 1.1, il peut sembler perturbant que par exemple une application avec des chances de défaillance très faibles (E1), mais où les répercussions sont très graves (S3) pourra être d'un niveau de criticité faible (QM ou ASIL A selon la contrôlabilité). Inversement, une application avec peu, voire aucune, conséquence en cas de faute (S1) mais un risque de défaillance très élevé (E4) peut avoir le même niveau de criticité (QM, A ou B selon la contrôlabilité). La prise en compte de la probabilité d'occurrence ne semble donc pas bien correspondre avec la définition que l'on se donne de la criticité qui va principalement dépendre de la Sévérité et de la Contrôlabilité. Alors même que ce genre de solution est courant dans les mécanismes de sûreté de fonctionnement de systèmes à criticité mixte, mais avec la première définition suscitée de la criticité.

De plus, les niveaux de criticité dans le domaine industriel impliquent souvent des propriétés de composabilité, qui permettent de combiner des composants à plus faible niveau de criticité pour obtenir l'équivalent d'un unique composant de niveau de criticité supérieure. Par exemple, si un capteur automobile requiert pour son usage un niveau d'ASIL A, il est possible d'utiliser conjointement deux capteurs avec un niveau d'ASIL B en équivalent. L'objectif des compositions étant de limiter les coûts, étant donné que plus le niveau d'ASIL est élevé, plus les exigences de développement et de vérification sont forts.

Enfin, que ce soit dans un cas comme dans l'autre, il est question de **Modes** de fonctionnement. Mais là encore cela peut impliquer des sens différents. Il existe d'une part les **Modes de Service**, c'est-à-dire des modes de fonctionnement dont l'objectif est uniquement la reconfiguration pour la survie et le bon fonctionnement du système. Il y a ensuite les **Modes d'Opération** qui décrivent des modes de fonctionnement dans l'usage "normal" du système. Par exemple des modes "décollage", "vol de croisière" et atterrissage" pour un avion. Et enfin les **Modes d'ordonnancement** qui se focalise exclusivement sur la façon selon laquelle le processeur va exécuter (ou non) les tâches pour permettre le bon ordonnancement global du logiciel.

Au regard de ces disparités, il convient donc de spécifier selon quel critère nous souhaitons définir les tâches *critiques* pour lesquelles nous désirons conserver des garanties de temps d'exécution pire cas. L'objectif inclusif de ces travaux étant de limiter pour une application donnée les risques de dépassement d'échéances d'exécution du logiciel qui réalise le service essentiel du système, les définitions se veulent volontairement générales pour pouvoir s'adapter selon les besoins du cas d'étude. Cette notion de criticité liée à des cas réels d'application a déjà pu être réfléchi sous le terme d'**importance** dans les travaux de [Fleming 2013], qui ont pu être approfondis par la suite par [Bletsas 2018] puis [Sundar 2019].

Définition 3.1.1. – Criticité

La criticité d'une tâche exécutée sur un processeur donné pour réaliser une fonctionnalité du système se définit selon son **importance**, notamment en termes de Sévérité et de Contrôlabilité. L'importance se mesure par les conséquences en cas de défaillance de cette fonctionnalité à la fois au regard de l'utilisateur (dangers) ainsi que l'importance de la fonctionnalité vis-à-vis des cas d'application essentiels du système, il s'agit de la Sévérité. Tandis que l'importance est diminuée à mesure des solutions de contrôlabilité existantes pour compenser les défaillances de cette fonctionnalité.

Le système que nous allons étudier ici est dit à niveau de criticité dual. Il exécute un ensemble de tâches logicielles (aussi appelée *charge utile*) exécutées sur un support logiciel (classiquement, le système d'exploitation). Elles se répartissent entre les tâches à haute criticité d'une part (dites Critiques, par la définition donnée en 3, et à faible criticité d'autre part (non critiques). Dans ce cadre particulier, il est alors possible de définir :

Définition 3.1.2. – Tâche critique

Une tâche critique est une tâche au niveau d'importance élevée. On peut aussi parler communément de tâche "vitale", dans le sens où une défaillance de cette fonctionnalité aurait soit des conséquences sévères pour l'utilisateur, soit empêcherait le bon fonctionnement d'un des modes d'opération essentiels du système. Inversement, une tâche pour laquelle une faute provoquant un dépassement d'échéance n'aurait pas de répercussions sévères sur l'utilisateur ou ne préviendrait pas la réalisation des fonctionnalités essentielles peut se définir comme non critique. En d'autres termes, une tâche de faible importance (relativement aux tâches critiques).

Enfin, en cohérence avec les modes de criticité mentionnés plus haut, nous définissons deux Modes de Service directement reliés à des Modes d'Ordonnancement pour contribuer à la sûreté de fonctionnement du système. D'une part le **Mode Nominal** où tout le système fonctionne normalement en l'absence de fautes. D'autre part, le **Mode Dégradé** dans lequel le système ne réalise pas la totalité des fonctionnalités de façon à pouvoir conserver des garanties d'exécution sur les tâches critiques et donc prévenir les défaillances catastrophiques, dans le pire des cas.

3.1.2 Modèle de Tâche et Chaînes de tâches

Maintenant que nous avons clairement posé la notion de criticité sur laquelle nous nous focalisons, il est possible d'expliciter notre objectif pour répondre à la problématique.

Rappelons ici que nous souhaitons exploiter au maximum les ressources matérielles disponibles au sein d'un multicoeur soumis à la gestion de cache. Dans le même temps, la contrainte propre aux fonctionnalités critiques du système nous impose de conserver des garanties de sûreté de fonctionnement de façon à éviter toute défaillance catastrophique. Pour répondre à cela, nous nous plaçons alors dans le cadre d'un système où les tâches sont divisées entre deux niveaux de criticité. D'une part les tâches non critiques qui sont potentiellement moins restreintes en

terme d'usage de ressources (temps de calcul et ressources partagées). D'autre part les tâches critiques suivant le critère d'importance susmentionné pour lesquelles on doit garantir une qualité de service avec une vision bout-en-bout. Ces éléments sont résumés sur la Figure 3.1.

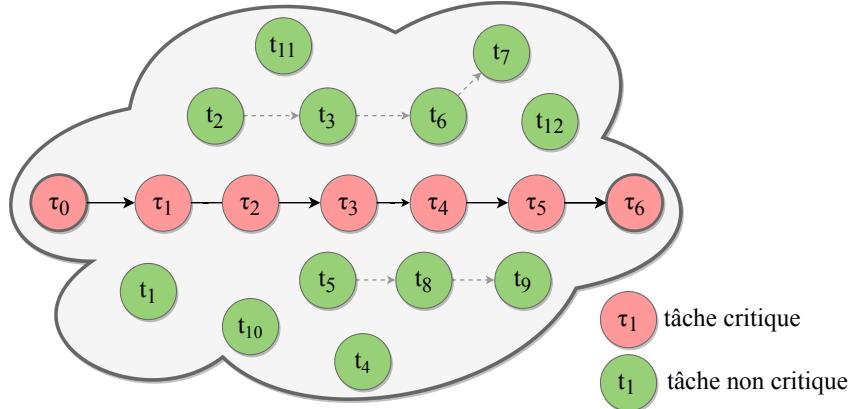


FIGURE 3.1 – Représentation simplifiée du set de tâches à criticité duale

3.1.2.1 Modèle de tâches

La plupart des hypothèses faites ici se focalisent sur les tâches critiques, tandis que la seule hypothèse forte sur les tâches non critique est la capacité à les stopper (soit un arrêt total, soit une mise en pause) et les relancer en cours d'exécution de façon à pouvoir déclencher un Mode Dégradé où il n'y a plus de tâches non critiques avec les risques d'interférences afférents envers les tâches critiques. Sous les systèmes type Unix, cela correspond typiquement à l'envoi d'un signal SIGSTOP et SIGCONT. Sans cette condition, les Modes de Service mentionnés ci-dessus ne sont pas exploitables pour notre besoin.

Chaque tâche critique τ_i est activée et exécutée suivant une période T_i . À chaque période, le job $\tau_{i,j}$ correspond à la $j^{ième}$ exécution de la tâche τ_i . On peut alors noter pour chaque job $\tau_{i,j}$ son moment d'activation $a_{i,j}$, son début d'exécution $s_{i,j}$ et sa terminaison $e_{i,j}$. On considère qu'un job consomme toutes ses données d'entrée (inputs) au début de son exécution, s'exécute et fourni à la fin de son exécution les données de sortie. Les données d'entrée et de sortie des tâches sont stockées en espace mémoire partagé : la transmission des données d'une tâche à l'autre se fait de façon asynchrone. Cela nous mène à la question de l'interaction entre les tâches et notamment la façon de représenter la précédence.

3.1.2.2 Chaînes de tâches

La question de la dépendance entre les tâches est importante pour aborder le problème des contraintes temps-réel avec une vision plus macroscopique. En effet, dans le cadre de l'usage de tâches ayant des contraintes temporelles souples (c.f. section 1.2.2 - Systèmes temps-réel), c'est uniquement avec une vision plus globale de l'exécution du système qu'il est possible de tirer au maximum parti des

légers dépassements pour éviter dans la globalité d'avoir recours à des politiques d'exécution des tâches plus restrictives, et par conséquent qui sous-exploite la puissance de calcul disponible. Nous considérons ici la dépendance entre les tâches via les données partagées entre ces dernières selon un modèle type producteurs/-consommateurs. Les tâches ont des relations de cause à effet et par conséquent, d'un point de vue strictement fonctionnel on peut décrire le système comme étant une accumulation de fonctionnalités réalisées par l'exécution de tâches successives. Cela permet alors d'introduire la notion de contrainte temporelle fonctionnelle, qui décrivent des contraintes d'exécution de chaînes de tâches bout-en-bout.

On représente une dépendance entre tâches sous la forme de chaînes de tâches, suivant le modèle $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n$. Dans un tel exemple, τ_1 est la **tâche d'entrée** de la chaîne, tandis que τ_n est la **tâche de sortie** de la chaîne. Notons que ce modèle peut être étendu pour supporter des tâches représentées par un Diagramme Orienté Acyclique (*Directed Acyclic Graph - DAG*) sans difficulté. Nous travaillons dans le cadre de ces travaux avec des chaînes directes, sans divergences ou convergences dans le graphe. De fait, cet ajout de complexité dans le modèle de chaîne de tâches n'apportera pas de différences particulières sur les résultats ni sur la démarche. De fait la différence fondamentale étant sur la façon de monitorer et reconstituer l'état de la chaîne de tâches, ce sont des éléments qui sont modifiables simplement pour prendre en compte un DAG complet. Le reste du principe présenté demeure fondamentalement inchangé.

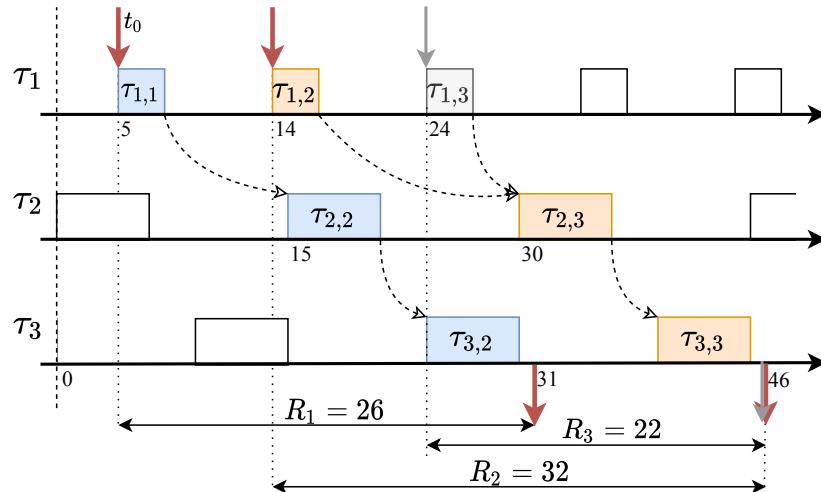


FIGURE 3.2 – Exemple d'exécution d'une chaîne de tâches $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$

Dans ce contexte, on définit à l'exécution la précédence entre les tâches par le biais d'un lien type producteur-consommateur. On considère que pour une tâche, les données produites sont disponibles à partir de la terminaison de cette dernière. Chaque tâche dans la chaîne de tâche consommera les données de sortie de tous les jobs de la tâche précédente dans cette chaîne qui n'ont pas encore été consommées au début de son exécution. En regardant l'exemple de la Figure 3.2 avec une chaîne $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$, à considérer que chaque job est désigné par la notation $\tau_{i,j}$, i indiquant

la tâche et j l'indice d'occurrence d'exécution de cette tâche. On peut voir que le job $\tau_{2,2}$ consomme les données de $\tau_{1,1}$, mais pas celles de $\tau_{1,2}$, étant donné que ce job n'est pas encore terminé à $t=15$. Aussi, $\tau_{2,3}$ pourra consommer les données de $\tau_{1,2}$ et $\tau_{1,3}$ étant donné qu'elles n'ont pas été consommées précédemment.

De façon formelle, on peut définir cette relation entre une tâche τ_i et son successeur τ_{i+1} . Pour produire la donnée de sortie du job $\tau_{i+1,k}$ de la tâche τ_{i+1} , ce dernier consomme toutes les données d'entrée en attente provenant des jobs $\tau_{i,j}$. Les données en attente étant celles qui n'ont pas été consommé par le job $\tau_{i+1,k-1}$ qui est le job précédent de τ_{i+1} . On peut donc écrire que pour un $\{i, k\}$ donnés, sont consommés les données de tous les jobs $\tau_{i,j}$ tel que $e_{i,j} \leq s_{i+1,k}$ et $e_{i,j} > s_{i+1,k-1}$. Autrement dit, un job $\tau_{i,j}$ n'a un effet sur $\tau_{i+1,k}$ si et seulement si ce dernier est le premier job de τ_{i+1} exécuté après la terminaison de $\tau_{i,j}$.

Remarque de compréhension : *les indices j et k ci-dessus représentent tous deux la différenciation des jobs pour une tâche donnée (τ_1 pour j et τ_2 pour k). Si 2 lettres différentes sont utilisées, c'est uniquement du fait qu'il n'y a pas de relation entre 2 jobs de tâches différentes qui auraient le même indice d'occurrence. D'où la distinction j/k .*

Dans ces conditions, on nomme $\tau_{i+1,k}$ le **successeur** du job $\tau_{i,j}$. On note $\text{succ}()$ la fonction qui permet de trouver le successeur d'un job donné tel que :

$$\forall i, j \in \mathbb{N}^*, \exists k \in \mathbb{N}^* \text{ tel que}$$

$$\text{succ}(\tau_{i,j}) = \tau_{i+1,k} \text{ avec } e_{i,j} \leq s_{i+1,k} \wedge e_{i,j} > s_{i+1,k-1}$$

Par extension, la fonction itérative $\text{succ}^p()$ pour tout $p \geq 0$ permet de trouver le $p^{\text{ème}}$ job successeur d'une tâche donnée de la chaîne de tâches. Et in fine, $\text{succ}^{n-1}(\tau_{1,j})$ désigne le job de sortie de la chaîne de tâche désignée par son job d'entrée $\tau_{1,j}$. On notera que pour $p = 0$, on définit $\text{succ}^0(\tau_{i,j}) = \tau_{i,j}$, soit la fonction identité.

Pour illustrer cela, on peut reprendre l'exemple de la chaîne représentée en Figure 3.2. On peut par exemple voir qu'une des exécutions de la chaîne de tâche, débutant par $\tau_{1,1}$, donne : $\text{succ}^2(\tau_{1,1}) = \text{succ}(\text{succ}(\tau_{1,1})) = \text{succ}(\tau_{2,2}) = \tau_{3,2}$. Mais aussi que le job $\tau_{2,3}$ doit prendre en compte les valeurs de sorties de 2 jobs de la tâche τ_1 . Plus précisément, $\text{succ}(\tau_{1,2}) = \text{succ}(\tau_{1,3}) = \tau_{2,3}$. En effet, étant donné que les tâches peuvent être définies par des périodes d'activation différentes, cela signifie notamment que si une tâche τ_i est exécutée plus fréquemment que son successeur $\text{succ}(\tau_{i+1})$, alors il est possible qu'un job $\tau_{i+1,j}$ soit le successeur de plusieurs jobs de la tâche τ_i .

Par extension et commodité d'usage, on ajoutera comme notations utiles $\text{succ}(a_{i,j})$, $\text{succ}(s_{i,j})$, $\text{succ}(e_{i,j})$ respectivement les temps d'activation, de début d'exécution et de terminaison du successeur du job $\tau_{i,j}$. Et de même par fonction itérative, $\text{succ}^k(s_{i,j})$ la date de début du $k^{\text{ième}}$ successeur de la tâche $\tau_{i,j}$:

Soit $\text{succ}(\tau_{i,j}) = \tau_{i+1,k}$, alors $\text{succ}(a_{i,j}) = a_{i+1,k}$

$$\text{succ}(s_{i,j}) = s_{i+1,k}$$

$$\text{succ}(e_{i,j}) = e_{i+1,k}$$

De même, si $\text{succ}^n(\tau_{i,j}) = \tau_{i+n,k}$ alors $\text{succ}^n(a_{i,j}) = a_{i+n,k}$

$$\text{succ}^n(s_{i,j}) = s_{i+n,k}$$

$$\text{succ}^n(e_{i,j}) = e_{i+n,k}$$

Cette façon de considérer les choses permet une plus grande flexibilité de notre modèle pour s'adapter à un cas concret. De cette façon le modèle gère déjà un nombre assez significatif d'implémentations de tâches existantes :

- Des tâches avec une file d'attente en entrée : lorsque la tâche est exécutée elle consomme toutes les données d'entrée en attente – le cas de base que nous considérons.
- Des tâches qui ne consomment que la donnée d'entrée la plus récente, les données précédentes étant considérées obsolètes. Dans ce cas-là, c'est équivalent à considérer que toutes les données précédentes sont consommées, mais qu'uniquement la plus récente est prise en compte, celle des jobs plus anciens sont ignorées.
- Pour les tâches où chaque exécution de la tâche ne prend en compte qu'une seule donnée en file d'attente, notre modèle ne gère pas tous les cas. Cela peut être la donnée la plus ancienne (stratégie FIFO¹) ou la plus récente (stratégie LIFO²). Quoi qu'il en soit, notre capacité à prendre en compte ce type de tâche va être dépendant de leur fréquence d'activation.
 1. Si les tâches s'exécutent toutes à la même fréquence, alors il y aura systématiquement une relation 1:1 entre producteurs et consommateurs.
 2. Si les tâches successeur dans une chaîne donnée s'exécutent systématiquement à une plus grande fréquence que les tâches précédentes, alors nous sommes aussi dans un cas avec jamais plus d'une donnée disponible en file d'attente. En effet, la tâche Consommateur s'exécutant plus rapidement que la Tâche productrice, le ratio Production/Consommation est inférieur à 1.

Ces 2 cas sont donc équivalents au premier type de tâches mentionné pour lequel toutes les données produites sont consommées, mais où la quantité de données en attente est systématiquement inférieure ou égale à 1 en réalité.
- 3. En revanche si une tâche successeur dans une chaîne s'exécute plus lentement que sa prédécesseure, c'est le cas qui n'est pas géré par ce modèle. En effet, si une seule donnée est consommée par job, et le reste stocké

1. FIFO : First-In First-Out, les données sont traitées de la première arrivée à la dernière.

2. LIFO : Last-In First-Out, les données sont traitées de la dernière (plus récente) arrivée à la plus ancienne.

en file d'attente, alors il pourrait y avoir un ratio Production/Consommation supérieur à 1. C'est-à-dire une plus grande quantité de données produites que consommées. Ce cas est relativement improbable (à moins d'une erreur de conception), car cela implique que la quantité de données en file d'attente peut potentiellement diverger. Or la file d'attente ne peut être considérée infinie.

De façon succincte, quand on parle de consommer plusieurs jobs de la tâche précédente tel qu'on le présente ici, cela n'implique pas que toutes ces données seront prises en compte. Tout dépend du modèle de fonctionnement interne des tâches. Mais quoi qu'il en soit, cela permet de gérer un grand nombre de comportements classiques.

Nous ne considérons pas dans notre modèle d'éventuels délais entre l'instant où une tâche produit sa donnée de sortie et le moment où cette donnée est réellement disponible pour la ou les tâches consommatrices. Il faudrait pour cela ajouter dans notre modèle de précédence une constante de latence correctement estimée selon l'implémentation de la transmission des données entre les tâches. On pourra noter sur ces aspects les travaux de [Friese 2018] qui ont pu aborder la prise en compte de latences dans différents modèles de chaînes de tâches.

Temps de réponse bout-en-bout La notion de successeur permet de définir le temps de réponse bout-en-bout R_j de la j -ème instance d'exécution d'une chaîne de tâche. Ainsi R_j désigne le temps écoulé entre l'activation du *job* d'entrée $\tau_{1,j}$ de la chaîne, jusqu'à la terminaison du *job* de sortie $\tau_{n,k} = \text{succ}^{n-1}(\tau_{1,j})$. On a alors $R_j = \text{succ}^{n-1}(e_{1,j}) - a_{1,j}$ que l'on peut retrouver dans l'exemple du Figure 3.2. Sur cet exemple, il est possible de reconstituer trois instances d'exécution de la chaîne de tâches avec les trois temps de réponse correspondants : R_1 , R_2 , R_3 .

Intuitivement, l'échéance bout-en-bout représente alors la durée maximale acceptable pour qu'une donnée d'entrée de la chaîne ait un effet du côté de la sortie. Pour une fonctionnalité donnée on comprend bien que cette échéance doit être bornée, et qu'il faut donc des garanties pour que tout se passe bien de bout-en-bout. Lorsque l'on considère l'échéance d'une chaîne de tâches D_c , pour éviter toute faute temporelle de non-respect d'échéance, il faut à minima respecter : $\max_{j \in \mathbb{N}} \{R_j\} \leq D_c$.

L'objectif à présent est de proposer une approche qui permette justement d'exploiter ces contraintes bout-en-bout, de façon à éviter les risques de fautes temporales au niveau fonctionnel. Cela se traduit par la volonté de prévenir les risques de dépassement d'échéances, non pas au niveau de chaque tâche individuelle, mais plutôt à un niveau d'observation au-dessus qui est en lien direct avec la représentation fonctionnelle.

3.2 Mécanisme d'anticipation par Surveillance et Contrôle

3.2.1 Méthode d'anticipation

Je propose donc un mécanisme basé sur la surveillance à l'exécution de l'avancement d'une chaîne de tâche. Pour ce faire, on introduit les notions d'**État de Chaîne de Tâche** et de **Trace d'Exécution de Chaîne de Tâche**. Une chaîne de tâches donnée est associée à un État et plusieurs Traces d'Exécutions. Ces deux éléments évoluant au fil de l'exécution du système, on peut noter $S(t)$ l'État à l'instant t et $ET(j, t)$ la $j^{\text{ème}}$ Trace d'Exécution de la Chaîne de tâches à l'instant t .

On peut alors définir pour une Chaîne de Tâches dont la tâche d'entrée est τ_1 , et la tâche de sortie τ_n :

Définition 3.2.1. Une **Trace d'Exécution** $ET(j, t)$ se définit par le début d'exécution d'un job d'entrée $s_{1,j}$ ainsi que tous les temps de fin d'exécution $e_{i,j}$ des *successeurs* itératifs de ce job qui ont été identifiés à l'instant t .

$$ET(j, t) = \{ a_{1,j}, e_{1,j}, \text{succ}(e_{1,j}), \dots, \text{succ}^{n-1}(e_{1,j}) = e_{n,k} \}$$

Une Trace d'Exécution $ET(j, t)$ est dite *active* si son job de départ $\tau_{1,j}$ a déjà été activé à l'instant t , et que son successeur itératif correspondant à la tâche de sortie de la chaîne $\tau_{n,k}$ n'a pas encore été terminé. Autrement, elle est *inactive*. En d'autres termes :

$$ET(j, t) \text{ est active ssi } a_{1,j} \leq t \text{ et } e_{n,k} > t$$

Il est possible de définir à partir des Traces d'exécution, actives ou inactives à un instant t , l'État de la Chaîne de tâches :

Définition 3.2.2. L'**État d'une Chaîne de Tâches** $S(t)$ défini à un instant t l'état d'avancement de l'exécution de la chaîne de tâches qui est toujours en cours et qui a été activée la plus anciennement. Autrement dit, $S(t) = \langle t_0, \tau_p \rangle$ basé sur la Trace d'Exécution la plus ancienne active $ET(j, t)$ tel que :

$$\begin{cases} t_0 &= \min_{j \in \mathbb{N}^*} \{ a_{1,j} \mid a_{1,j} \leq t \wedge \text{succ}^{n-1}(e_{1,j}) > t \} \\ \tau_p = \text{succ}^p(\tau_{1,j}) &= \min_{l \in \llbracket 0; n-2 \rrbracket} \{ \text{succ}^{l+1}(\tau_{1,j}) \mid \text{succ}^l(e_{1,j}) \leq t \} \end{cases}$$

Avec t_0 la plus ancienne activation parmi les $ET(j, t)$ et τ_p la prochaine tâche de cette trace d'exécution qui n'a pas encore été terminée.

De cette façon, l'État d'une chaîne de tâches indique quelles sont les tâches restantes à exécuter dans la chaîne à un instant donné et son temps de réponse partiel actuel que l'on notera $RT(t) = t - t_0$. Et l'état de la chaîne se focalise sur la chaîne active la plus ancienne, donc la plus longue ce qui implique que c'est la plus à même de dépasser l'échéance.

Pour finir, au regard de l'État d'une chaîne de tâches, on peut s'intéresser au temps restant jusqu'à la complétion de cette chaîne. Il est possible d'estimer le

temps qu'il faudra pour aller jusqu'à exécuter la tâche de sortie de la chaîne qui fait partie de la Trace d'Exécution active observée, c'est-à-dire $\text{succ}^{n-1}(\tau_{1,j}) = \tau_{n,j}$. Et si, de plus, cette estimation est faite dans les hypothèses d'estimation de Temps d'Exécution Pire Cas, on obtient alors une estimation de Pire Temps de Réponse restant $rWCRT(t)$ à l'instant t pour terminer l'exécution de la trace d'exécution active la plus longue.

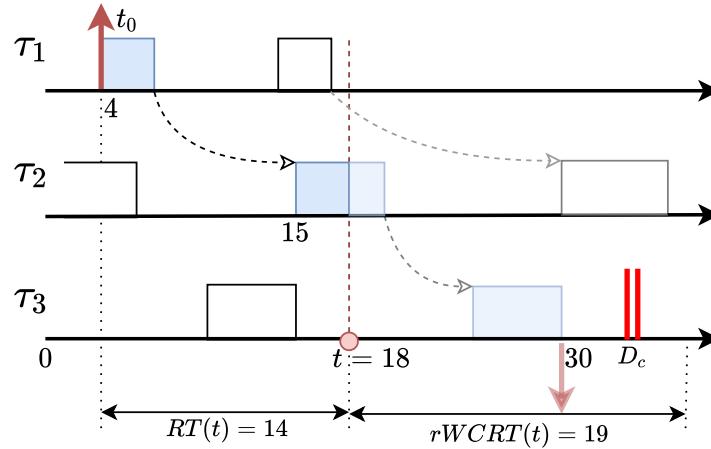
Alors, en combinant l'État à un instant t avec une estimation du Temps de Réponse Pire Cas restant, on peut donc estimer une borne haute garantie de temps de réponse de la chaîne de tâche. C'est là que l'on peut faire entrer en jeu un mécanisme d'anticipation. On dispose pour une chaîne de tâches donnée de son temps de réponse partiel actuel ainsi qu'une estimation de temps de réponse restant en pire cas. Il est alors possible de déterminer s'il y a un risque de défaillance par dépassement de l'échéance bout-en-bout D_c .

Théorème 3.2.3 (Risque de dépassement d'échéance). *Si à un instant donné t , l'inéquation suivante est respectée, alors il y a risque de dépassement d'échéance.*

$$RT(t) + rWCRT(t) > D_c$$

Pour illustrer cette logique, on peut voir sur le chronogramme 3.3 à nouveau un exemple avec une chaîne de tâches $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. À l'instant $t = 18$ d'indiqué il y a deux Traces d'Exécution actives (chaînes reliées par une flèche de succession). On a de ce constat l'État de la chaîne $S(t) = \langle \tau_0, \tau_2 \rangle = \langle 5, \tau_2 \rangle$. On en déduit $RT(18) = t - t_0 = 18 - 4 = 14$. Si l'on ajoute à cela une estimation du Temps de Réponse Pire Cas restant $rWCRT(18)$, qui est le temps estimé pour que $\rightarrow \tau_2 \rightarrow \tau_3$ soit exécuté selon les contraintes de précédence, alors on a l'estimation du Pire Temps de Réponse : $RT(18) + rWCRT(18) = 33$ que l'on peut comparer à la date d'échéance $D_c = 30$. Dans cet exemple, il existe donc un risque de dépassement de l'échéance. Il est à noter aussi que dans cet exemple l'instant t a été pris en plein pendant l'exécution de la tâche τ_2 . Ce qui est pris en considération comme si cette dernière n'était pas exécutée. Si l'on ne prend pas non plus en compte son exécution partielle, c'est parce que d'un point de vue externe à cette tâche, sans l'instrumentaliser il n'est pas possible de le savoir. Et de fait, l'une de nos contraintes étant d'être le moins intrusif possible sur le code, notamment pour les cas où certains logiciels ne sont pas modifiables (black-box ou legacy).

À présent, il faut nous souvenir de notre objectif dans tout cela. Pourquoi vouloir anticiper une défaillance de la sorte ? L'objectif est d'anticiper un risque de dépassement d'échéance à l'échelle d'une chaîne de tâches, de façon à pouvoir passer d'un Mode d'ordonnancement Nominal vers un mode Dégradé dans lequel on va prévenir la défaillance. Cette dernière a pour origine première les interférences matérielles qui augmentent les temps d'exécution des tâches. Aussi pour éviter davantage ce glissement temporel et donc conserver la garantie de terminaison de la chaîne de tâches avant l'échéance, on remonte à la source en prévenant temporairement tout risque d'interférences. Et cela est obtenu via un mode dégradé dans lequel la méthode consiste à stopper temporairement les tâches non critiques, cause des interférences. Il nous faut décrire à présent la méthode de passage en mode dégradé.

FIGURE 3.3 – Chronogramme de calcul du risque de dépassement à un instant t

3.2.2 Passage en Mode Dégradé

Estimation de Temps d'Exécution Pire Cas restant Bien évidemment, l'estimation du Temps de Réponse Pire Cas restant est un élément clé de l'approche. Tout l'intérêt de cette méthode réside dans la capacité à passer dans un mode dégradé. En conséquence, ce que nous nous devons de garantir, c'est le non-dépassement d'échéance sachant qu'il est possible à tout moment de prendre la décision du passage en mode dégradé, dans lequel les tâches critiques n'étant plus sujettes à interférences externes, auront un Temps d'Exécution Pire Cas bien plus faible qu'en mode Nominal. Cela implique directement que si le $rWCRT(t)$ que nous considérons est dans le contexte Dégradé, alors la détection d'un éventuel risque se fait de façon beaucoup plus permissive que si l'on considère directement les risques en mode Nominal. Étant donné que le mode dégradé correspond à un arrêt des tâches non critiques, on peut considérer que dans ce mode la chaîne de tâches est en isolation. De ce fait, la seule chose qui va modifier l'estimation du $rWCRT(t)$ va être l'État de la chaîne de tâches à l'instant t , $S(t)$. Or, étant donné que $S_c = \langle t_0, \tau_i \rangle$, alors on remarque qu'il n'y a que la valeur de τ_i – c'est-à-dire la prochaine tâche de la chaîne qui doit être exécutée – qui va influer sur la valeur estimée du pire temps de réponse restant. Ainsi, $rWCRT(t)$ peut être simplifié en une fonction discrète qui évolue avec $S(t)$. Par conséquent, on notera pour la suite de notre raisonnement $rWCRT(t) = rWCRT(\tau_i)$.

Ceci étant posé, il existe plusieurs méthodes à l'estimation de $rWCRT(\tau_i)$. De façon théorique, il est possible d'exploiter les méthodes déjà existantes d'estimation de temps d'exécution pire cas, auxquelles il faut ajouter la prise en compte des temps d'activation des tâches. Ce type d'approche devient hautement dépendante du système étudié, que ce soit l'architecture matérielle, mais surtout la politique d'ordonnancement des tâches, le type de tâches (périodique, sporadique, interruption)... De façon générale, la complexité des approches théoriques n'est pas négligeable et, il faut l'admettre, hors de notre cadre d'expertise. C'est d'autant plus vrai dans un cas d'application sur processeur multicœur pour lequel il est facile de tomber dans

des estimations trop pessimistes. Pour cette raison, nous avons décidé dans notre proposition d'avoir une approche plus expérimentale.

Un avantage dont nous bénéficions ici, c'est que l'hypothèse de se ramener à un cas en isolation dans le mode dégradé limite grandement les risques de variations sur les temps d'exécutions des tâches critiques. C'est ce qui permet une plus grande certitude sur les estimations expérimentales, qui ne nécessitent alors plus de couvrir toute une combinatoire incluant les tâches non critiques. L'estimation est donc faite expérimentalement en exécutant le système avec un passage forcé en mode dégradé dans lequel on peut alors mesurer sans les tâches non-critiques les Temps de Réponse Pire Cas restants $rWCRT(\tau_i)$. On notera que pour une chaîne de N tâches, il y a N-1 $rWCRT$ à estimer. Plus de détails sur le protocole adopté pour l'estimation seront abordés en chapitre 4.

La transition en elle-même vers le mode dégradé est une phase importante du fait qu'elle implique des délais supplémentaires qui devront être pris en compte dans l'anticipation. De cette façon, à considérer que l'on recalcule périodiquement le risque de dépassement d'échéance, il est possible de définir l'instant où l'on sait que l'attente d'une période supplémentaire va faire que même en mode dégradé, il ne sera plus possible de garantir le respect de l'échéance bout-en-bout. Par conséquent, il devient alors clair que cet instant-là devient le dernier moment auquel il faut nécessairement passer en mode dégradé pour justement conserver cette garantie.

Transition de Mode Ce changement de mode se fait en 3 étapes. Il faut en premier lieu bien entendu la détection du point de bascule auquel le risque de défaillance est détecté, c'est l'étape de décision. Une fois la décision prise, la seconde étape est d'activer le mécanisme de passage en mode dégradé. Dans notre cas, il s'agit d'envoyer un signal aux tâches non critiques de façon à mettre en pause leur exécution, c'est l'étape de contrôle. Enfin, le système de surveillance doit continuer d'observer l'État de la chaîne de tâches de façon à relancer les tâches non critiques une fois le risque passé. Il s'agit de l'étape de recouvrement.

Ces trois étapes font ressortir un élément important pour l'anticipation qui n'a pas été pris en compte pour le moment, et il s'agit de la durée entre l'étape de décision et la fin de l'étape d'exécution. En effet, le temps pour que toutes les tâches non critiques soient mises en pause est non nul, et ce délai doit être pris en compte dans l'estimation de Temps d'Exécution Pire Cas restant.

En conclusion, en considérant les grandeurs suivantes :

- W_{MAX} La durée maximum garantie entre 2 points de surveillance de l'Etat de la chaîne de tâche
- t_{SW} Le délai maximum nécessaire à la transition du mode nominal au mode dégradé
- $rWCRT(\tau_i)$ pour chaque τ_i de la chaîne de tâche, les Temps de Réponses Pire Cas restant en mode dégradé

Il est alors possible de calculer la somme du temps de réponse partiel actuel avec ces trois métriques. Tant que cette somme reste inférieure à l'échéance bout-en-bout, alors on peut conserver le système en mode nominal. En revanche, à partir du moment où cela dépasse l'échéance, alors c'est le moment où il n'est plus sûr de

rester en mode nominal, et il faut donc déclencher le mode dégradé pour garantir le respect de l'échéance. Cela correspond en conséquence à surveiller que l'inéquation suivante reste vraie pour savoir l'instant critique auquel il faut passer en mode dégradé :

$$RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW} \leq D_c \quad (3.1)$$

Cette équation est notamment adaptée des travaux de [Kritikakou 2014]. Chaque point de surveillance de l'État de la chaîne de tâches est considéré comme étant temporellement sûr (au sens où il n'y a pas de risque de faute temporelle due au dépassement d'échéance) tant que cette inégalité est respectée.

Démonstration. En présumant que (3.1) est respectée, on peut montrer qu'il est sûr d'attendre le prochain point de surveillance pour décider de changer de mode.

À l'instant de surveillance t , soit t_{next} le prochain instant de surveillance.

Par définition, $t_{next} \leq t + W_{max}$. Alors $RT(t_{next}) \leq RT(t) + W_{max}$. Par conséquent,

$$RT(t_{next}) + rWCRT(\tau_i) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}.$$

Aussi, $rWCRT()$ ne peut que décroître avec le temps qui s'écoule. Par conséquent,

$$rWCRT(t_{next}) \leq rWCRT(\tau_i)$$

et

$$RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$$

Étant donné que l'inégalité (3.1) est respectée, on a $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq D_c$. De ce fait, il sera sûr de passer en mode dégradé au prochain point de surveillance. \square

En conclusion, grâce au calcul d'anticipation de risque de dépassement d'échéance, il nous est possible d'identifier l'instant t auquel il est nécessaire de passer en Mode Dégradé. La démarche que suit notre mécanisme d'anticipation est illustré en Figure 3.4 avec l'hypothèse d'une chaîne de tâche :

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4 \rightarrow \tau_5 \rightarrow \tau_6$$

À chaque point de Surveillance, le mécanisme anticipe le risque en calculant l'inégalité 3.1 comme reconstitué en Figure 3.4a. Si l'inégalité n'est plus respectée et donc que le risque est manifeste comme sur le chronogramme, alors le passage en mode dégradé est effectué, de façon à ce que ce soit le scénario représenté en Figure 3.4b qui se réalise, et donc éviter le dépassement d'échéance potentiel. On notera la présence de la tâche non critique t_0 qui s'exécute sur le même cœur mais n'influe pas sur le calcul d'anticipation qui est fait.

3.3 Architecture Logicielle

Maintenant que nous avons présenté toute la logique derrière le mécanisme de surveillance et de contrôle, nous allons présenter plus en détail l'architecture logicielle nécessaire à son implémentation.

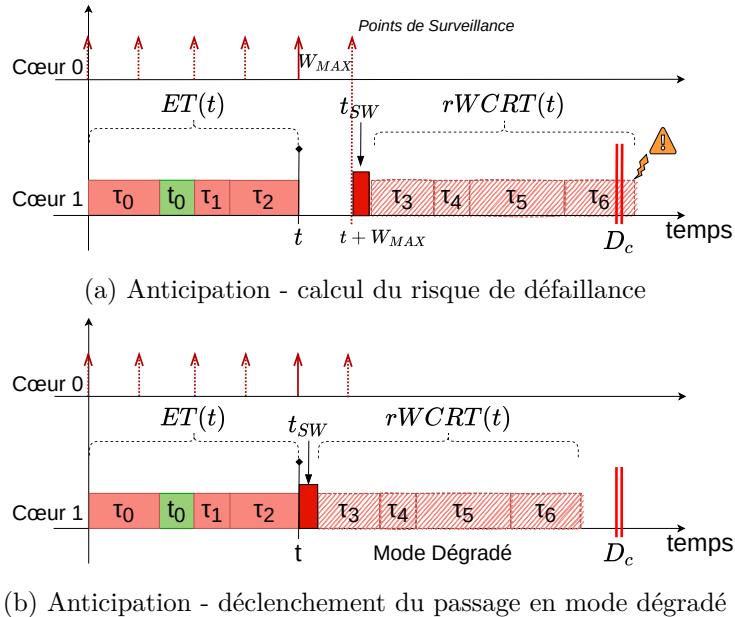


FIGURE 3.4 – Chronogrammes de fonctionnement du mécanisme d'anticipation

Pour résumer, nous sommes dans le cas d'un système à criticité mixte implémenté sur un calculateur multicoeur, dans lequel nous avons distingué une chaîne de tâches critiques des autres tâches considérées non-critiques. Nous souhaitons ajouter un Agent de Surveillance et de Contrôle pour assurer la fonctionnalité de mitigation des fautes temporelles sur la chaîne de tâches critique. Ce dernier se destine à être exécuté en couche bas niveau, au même niveau que la politique d'ordonnancement du système. Dans le cadre de la suite de nos expérimentations, pour simplifier l'implémentation, nous avons considéré l'isolation de l'Agent de Surveillance et Contrôle sur un cœur du processeur. L'ensemble de ces éléments se résume en Figure 3.5.

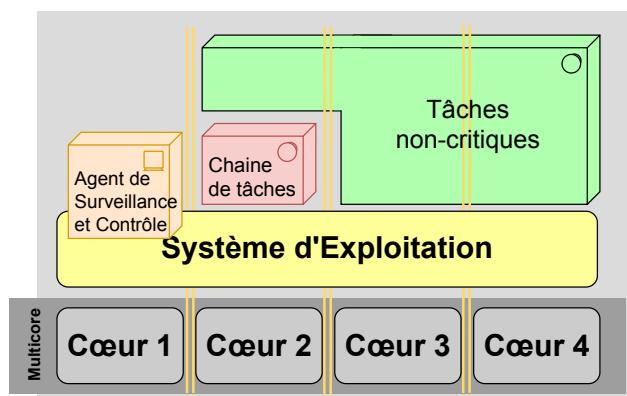


FIGURE 3.5 – Architecture Conceptuelle du mécanisme de Surveillance et Contrôle

L'Agent de Surveillance et Contrôle peut se décomposer en deux éléments distincts. D'un part un *Task Wrapper Components* (TWC) et de l'autre le *Core Control*

Component (CCC). Le TWC se destine à récupérer toutes les informations nécessaires à la surveillance et la mise à jour de l'État de la Chaîne de tâches par le biais d'une encapsulation des tâches , tandis que le CCC doit prendre en compte ces informations, de façon à réaliser la prise de décision du passage en mode dégradé au regard de l'inéquation 3.1. Les interactions entre les différents composants logiciels sont représentées sur la Figure 3.6.

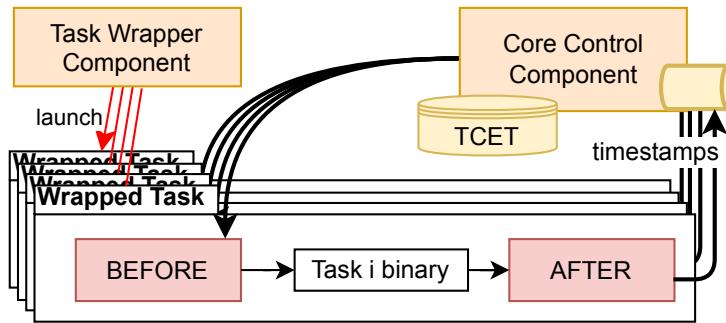


FIGURE 3.6 – Architecture Logicielle de l'Agent de Surveillance et Contrôle

3.3.1 Task Wrapper Component (TWC)

L'objectif du Task Wrapper Component est de gérer l'aspect Surveillance du mécanisme. Il englobe l'instrumentation du logiciel de façon non intrusive pour obtenir les informations nécessaires à la Surveillance. C'est une différence assez importante avec bon nombre de travaux qui se basent sur la surveillance de l'exécution de tâches. Souvent ces dernières reposent sur une instrumentation logicielle au niveau du code des tâches. Cela offre un suivi plus fin, mais avec un coût de développement logiciel supplémentaire pour chaque tâche. De plus c'est par définition incompatible avec du logiciel fourni en boîte noire.

Le mécanisme de Surveillance et Contrôle nécessite **a)** une connaissance des dates d'activation et de terminaison des jobs, ainsi que **b)** la connaissance de la structure de la chaîne de tâche. Concernant le premier point, il faut donc ajouter des blocs de code exécutés directement avant le début d'exécution des tâches ainsi que juste avant leur terminaison. En récupérant les *timestamps*³ d'exécution de ces blocs de code, il est ainsi possible de connaître précisément les dates de début $s_{i,j}$ et de fin $e_{i,j}$ de chaque job. Ils ont vocation à être envoyés au Core Control Component pour mettre à jour l'État de la chaîne de tâches ainsi que le suivi des Traces d'Exécution actives.

Par ailleurs, une telle instrumentalisation permet d'ajouter une couche de sécurité supplémentaire. En effet, le changement de mode se fait initialement par envoi d'un signal pour stopper les tâches non critiques. Ceci étant dit, pour une meilleure réactivité et ajouter de la redondance dans l'arrêt des tâches, il est aussi possible d'utiliser le bloc logiciel exécuté avant l'exécution des tâches non critiques pour

3. timestamp : en informatique, le temps se mesure par un compteur, une variable qui s'incrémente à la fréquence de fonctionnement du processeur. Un timestamp correspond à un horodatage sur ce compteur

vérifier si la tâche en question a effectivement l'autorisation de s'exécuter ou bien si on est en mode dégradé, doit être stoppée.

En somme, le Task Wrapper Component doit encapsuler les tâches pour ajouter deux wrappers : un **Before** et un **After** qui vont avoir deux rôles :

- envoyer au Core Control Component les timestamps d'exécution associés à chaque job de tâche critique (dates de début et de fin),
- ajouter de la redondance pour prévenir l'exécution des tâches non-critiques en mode dégradé et potentiellement accélérer le changement de mode.

On remarquera qu'il n'est pas utile d'exécuter un wrapper After à la suite des tâches non critiques étant donné que nous ne monitorons pas leur exécution dans le cadre du mécanisme.

3.3.2 Core Control Component (CCC)

Le Core Control Component gère donc l'aspect "Contrôle" du mécanisme. Il doit être exécuté périodiquement, tous les T_{CCC} unités de temps. À chaque exécution, son rôle est de récupérer toutes les données de timestamps du TWC. En ayant connaissance de la structure de chaîne de tâches qui est exécutée, il peut alors reconstruire les Traces d'Exécution telles que définies précédemment (Définition 3.2.1).

À partir des Traces actives, l'État de la chaîne de tâches $S(t)$ est alors mis à jour. Il s'agit de calculer successivement :

- $RT(t)$, la durée à partir de la date de départ de la tâche d'entrée,
- $rWCRT(\tau_i)$, le Temps de Réponse Pire Cas restant à prendre en compte selon l'état de la trace d'exécution active.

En y ajoutant les constantes W_{MAX} et t_{SW} , l'inégalité 3.1 est re-calculée. Dans le cas où elle est toujours respectée, rien ne se produit et le CCC peut attendre la prochaine période de vérification. Dans le cas où elle devient fausse, alors la procédure de changement de mode est enclenchée.

Il pouvait y avoir deux choix possibles d'activation du Core Control Component. Soit en le déclenchant de façon asynchrone, à chaque fois que la TWC reçoit une nouvelle donnée de timestamp, soit en le déclenchant de façon périodique indépendamment de l'exécution des tâches. Notre choix s'est porté sur la seconde option pour une raison fondamentale. En effet, bien qu'un déclenchement asynchrone soit plus simple à implémenter d'un point de vue technique, on perd la maîtrise de l'utilisation processeur de notre Agent de Surveillance et Contrôle, ce qui peut être dommageable pour la maîtrise de l'ordonnancement du système. De plus, dans le cas où une défaillance sur la réception des timestamps surviendrait, le CCC pourrait se retrouver dans une attente indéfinie de données, ne mettant alors pas à jour l'État de la chaîne de tâches... et laissant donc passer les risques de faute temporelle. Par conséquent, avec un déclenchement périodique, on obtient une décorrélation du reste du système. De ce fait, quoi qu'il arrive il sera possible à intervalle fixe de déterminer s'il y a un risque. En effet, même en l'absence de nouveaux messages la valeur de $RT(t)$ continue d'augmenter.

3.3.3 Définition des constantes de Contrôle

Il est important de fixer correctement les paramètres constants du CCC, qui sont t_{SW} et W_{MAX} . Si ces paramètres sont sous estimés, typiquement dans le cas où le temps de passage en mode dégradé est plus long, alors l'inégalité 3.1 pourra devenir tout bonnement fausse. Concernant la durée maximale entre 2 mises à jour de l'État de la chaîne de tâches W_{MAX} , pour une Surveillance périodique faite par une tâche avec le plus haut niveau de priorité, cela correspondra à la durée de la période fixée T_{CCC} . Là encore on réalise qu'avec une activation sporadique à l'arrivée des messages, cette durée aurait été plus complexe à fixer.

Cependant, cela repose sur le choix sur la fréquence d'exécution du CCC. Ce choix peut avoir plusieurs conséquences et va donc être sujet à de potentiels compromis. D'une part, un T_{CCC} trop petit signifie une plus forte utilisation des ressources de calcul, ce qui va directement à l'encontre de notre objectif d'optimiser la puissance de calcul. C'est d'ailleurs une problématique courante des mécanismes de Surveillance, d'avoir un impact négatif sur l'usage des ressources du processeur et s'avérer contre-productif.

À l'inverse, plus la période va être longue, plus le mécanisme d'anticipation sera sensible et anticipera des risques de plus en plus improbables. De fait, plus la prochaine date de vérification de l'État de la chaîne est éloigné dans le temps, plus le moindre glissement de temps d'exécution des tâches critiques va laisser penser à un risque de dépassement d'échéance. Cela fait alors augmenter le taux de faux positifs, c'est-à-dire le taux de déclenchements du passage en mode dégradé qui n'étaient pas nécessaires. Par ailleurs, il faut que le Core Control Component puisse soutenir le débit d'arrivée de données d'exécution des tâches critiques, car il n'est pas possible de maintenir en liste d'attente une quantité infinie de messages.

En conclusion, la valeur de T_{CCC} doit être choisie au regard d'une estimation du débit d'exécution des jobs, donc dépendant du nombre de tâches et de leur période d'exécution.

Une fois qu'un passage en mode dégradé a été effectué, le CCC doit veiller au recouvrement du système pour retourner en mode nominal en temps voulu. Les passages en mode dégradé se veulent les plus intermittents possibles étant donné qu'ils sont le fruit d'une anticipation pour éviter un potentiel dépassement d'échéance. L'objectif est donc de relancer au plus tôt les tâches non critiques. La stratégie choisie est de repasser en mode nominal à la terminaison de la chaîne de tâches qui a provoqué le passage en mode dégradé. Dans ce stade, il n'y a plus aucun risque présent de dépassement et la reprise de contexte des tâches mises en pause peut se faire normalement, sur leur prochaine période d'activation.

3.4 Application industrielle

3.4.1 Spécification fonctionnelle

Notre proposition d'Agent de Surveillance et Contrôle se veut assez générique et permissive en termes de contraintes d'implémentation de façon à pouvoir être adapté à un grand nombre de cas d'application. L'enjeu réside dans le choix d'une

combinaison appropriée de garanties temps-réel, et donc d'ajout des contraintes sous-jacentes, avec l'existant, et donc des composants logiciels qui ont déjà des spécifications données, voire un code qui n'est pas modifiable. Il semble alors approprié d'étudier à minima dans quelle mesure il est possible d'adapter ce mécanisme dans un contexte industriel. Nous allons voir dans ce contexte les différentes architectures logiciel dans l'automobile et l'avionique, ainsi que l'approche globale qu'il faudrait avoir pour approcher un mécanisme comme celui proposé.

Notre approche ne se focalise pas sur le respect des échéances temporelles individuelles des tâches, mais sur une vision plus macroscopique des contraintes bout-en-bout, l'objectif étant d'en obtenir une plus grande marge de manœuvre sur l'ordonnancement des tâches pour garantir ces contraintes. Cela suppose que nous nous plaçons a priori essentiellement sur des modèles de tâches critiques à temps-réel souple. Ainsi, il est possible qu'une tâche prenne un peu plus de temps qu'escompté pour être exécutée mais que cela soit potentiellement compensé par une tâche suivante qui rattrape ce délai. Il a été ainsi constaté qu'il arrive très rarement que toutes les tâches se retrouvent retardées simultanément. Au contraire, le ralentissement d'une tâche peut être dû fait qu'une autre a pris la priorité sur l'utilisation des ressources, et donc que cette dernière s'est, elle, exécutée correctement.

Une des questions principales dans le cadre d'un cas d'application réel réside dans la façon de passer des spécifications fonctionnelles aux tâches logicielles qui sont allouées et exécutées sur le processeur.

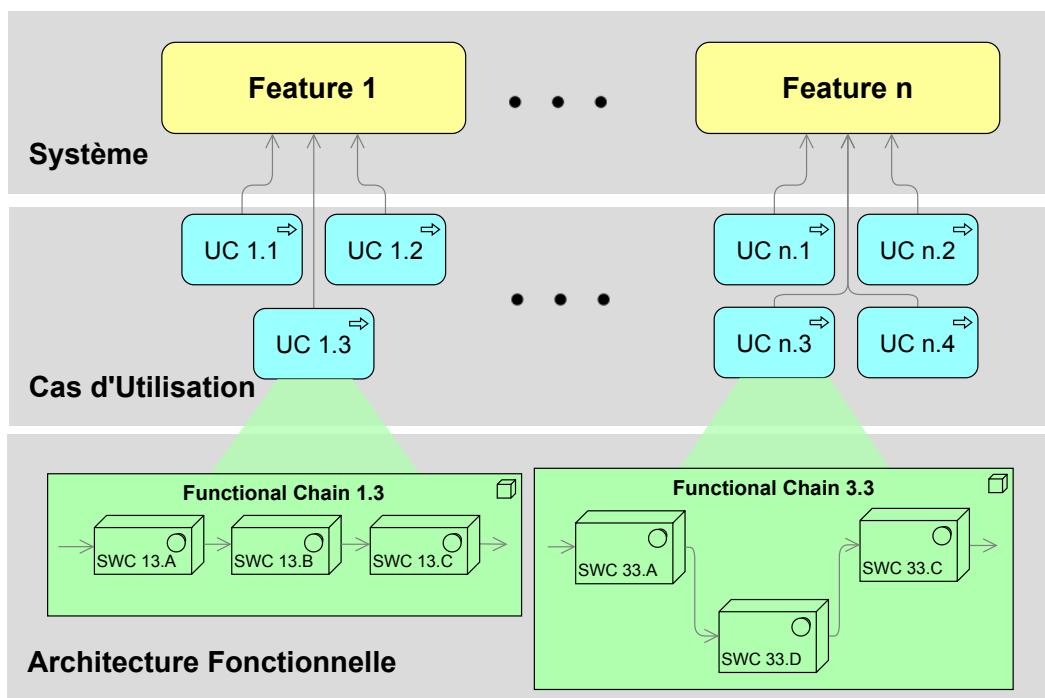


FIGURE 3.7 – Définition d'architecture fonctionnelle

Il faut considérer les chaînes de tâches comme une implémentation d'une fonctionnalité système, qui va de son déclenchement jusqu'à sa terminaison. Cela correspond

une majeure partie du temps à une chaîne de réaction allant d'une donnée d'entrée (détection d'un capteur par exemple) jusqu'à la conséquence (activation d'un actionneur), en passant par un algorithme de décision. Il s'agit donc d'un modèle classique de type Sense–Compute–Control où en pratique le Compute est implémenté par une Chaîne de tâches. Comme on l'a vu en début de ce Chapitre, il n'est pas tout à fait pertinent d'associer directement des niveaux de criticités au sens des standards logiciels de Sûreté de Fonctionnement (ASIL D à ASIL A et QM pour ISO 26262 en automobile typiquement), aux niveaux de criticités pour l'Agent de Surveillance et de Contrôle. Le seul lien qu'il est possible de déduire de la classification ASIL étant plutôt qu'il est improbable que les tâches de la chaîne critique soient de niveau QM. Les tâches de niveau d'ASIL élevé (ASIL D au maximum) feront certainement partie de la chaîne de tâches critiques, mais il est difficile à partir d'un niveau d'ASIL plus faible d'en déduire le niveau d'importance d'une tâche. Ce critère d'importance étant crucial étant donné que les tâches non critiques du système pourront être mises en pause par l'agent de Surveillance et Contrôle.

Si l'on revient au cas d'application automobile, les spécifications sont données par étapes. Dans un premier temps, il y a la spécification Système, qui liste la totalité des fonctionnalités (*Features*) qui doivent être embarquées. Chaque Feature contient un certain nombre de Cas d'Utilisation (*Use Cases*), qui se définissent par un scénario de fonctionnement et le comportement attendu de la fonctionnalité. Enfin, pour chaque Use Case, il est possible de représenter la Chaîne Fonctionnelle qui définit les blocs logiciels qui sont mobilisés pour réaliser ce cas d'utilisation. L'ensemble de ces chaînes constitue l'architecture fonctionnelle du système. Ces différentes couches de spécification système sont représentées dans la Figure 3.7.

On sent assez facilement que cette façon de modéliser le système est adaptée à notre approche pour identifier la méthode de sélection des tâches critiques, ainsi que quel type de spécification prendre en compte pour l'échéance d'exécution bout-en-bout, qui sont des données déjà existantes au niveau de l'architecture fonctionnelle. Une fois la chaîne fonctionnelle déterminée en fonction du système étudié, il est alors possible de descendre jusqu'à la spécification des tâches logicielles qui découlent cette architecture pour identifier les tâches critiques et non critiques.

CHAPITRE 4

Protocole et démarche expérimentale

Sommaire

4.1	Principe Général et Objectifs	62
4.2	Protocole de conception d'un cas de test	62
4.2.1	Profil des tâches	64
4.2.2	Profil du cas de test et Chaîne de tâches	66
4.3	Protocole d'Implémentation et Calibration	70
4.3.1	Spécification des paramètres de Contrôle	71
4.3.2	Calibration du Mécanisme de Contrôle	73
4.4	Protocole expérimental global	76

Jusqu'à présent, nous avons fait un état des lieux des enjeux liés à l'usage des calculateurs multicœurs dans le cadre d'applications à criticité mixte et en dégager une problématique. Après une revue des solutions existantes dans le domaine, nous avons présenté dans le Chapitre précédent nos hypothèses de travail ainsi que le modèle de notre mécanisme d'évitement de fautes temporelles basé sur la surveillance d'une Chaîne de tâches critiques. Nous allons maintenant aborder une méthodologie de mise en service et de calibration du mécanisme pour un cas d'application. Cela nous permettra d'identifier les fondamentaux qui serviront au cas d'étude du Chapitre suivant.

Notre proposition méthodologique repose sur 3 grandes phases de conception. Nous développerons successivement les exigences liées à la phase de Conception et de caractérisation d'un cas de test. L'objectif de cette démarche expérimentale est de caractériser un jeu de tâches au regard de sa propension à provoquer des interférences, et sa sensibilités aux-dites interférences.

Les étapes de calibration et d'ajustement du mécanisme ont pour but d'implémenter et de configurer le mécanisme de Surveillance et de Contrôle pour l'implémenter sur un système à criticité duale donné.

Enfin la phase de vérification consiste à mesurer les propriétés de notre mécanisme ainsi configuré, pour un système donné. Il s'agit de mesurer et d'analyser plusieurs métriques : l'Efficacité, la Fiabilité et la Qualité, au regard de la capacité à respecter l'échéance bout-en-bout de la chaîne de tâche, à limiter le temps d'arrêt des tâches non critiques, et à éviter les faux-positifs.

4.1 Principe Général et Objectifs

L'objectif de ce Chapitre est de présenter notre protocole expérimental, pour répondre à deux objectifs :

- Le premier de caractériser un système de tâches donné pour y définir une chaîne de tâches critiques ainsi que le profil temporel des différentes tâches.
- Le second de calibration et de test de l'Agent de Surveillance et Contrôle.

Il est important de préciser que notre méthode se décompose en ces deux intérêts bien distincts. En effet, une première partie du protocole général qui a été mis en place dans le cadre de ces travaux s'inscrit dans la phase de conception et de spécification. Cette partie se destine à la mise en place d'un banc expérimental qui ne repose pas sur un système déjà existant. De fait, en l'absence d'un système réel avec ses spécifications propres sur lesquelles se baser, il nous faut recréer un cas de test et donc en fixer les contraintes temporelles qui soient cohérentes avec les caractéristiques d'exécution du logiciel qui est utilisé. En conséquence, l'ajout d'étapes méthodologiques a été nécessaire pour donner des spécifications cohérentes à ce cas d'utilisation et connaître le comportement temporel de notre cas de test. Une fois cet aspect traité, la seconde partie du protocole expérimental présente normalement les étapes nécessaires qui doivent s'intégrer au processus de développement et d'intégration préexistant pour y ajouter l'Agent de Surveillance et Contrôle, le calibrer et le tester.

Ce chapitre se divise donc en deux sections qui correspondent aux processus susmentionnés. Ils permettront de définir en amont du développement et de l'intégration de la solution quelles sont les données dont nous disposons en entrée et quels sont les résultats attendus en sortie de chaque étape du processus. Dans un dernier temps nous agrégeront tout cela pour constituer un protocole expérimental global qui sera celui-utilisé pour la suite dans le cadre de notre mise en application.

4.2 Protocole de conception d'un cas de test

Le protocole de conception d'un cas de test proposé se divise en 5 phases distinctes. L'objectif est d'extraire à partir d'un set de tâches logicielles donné un jeu de tâches dont on connaît les profils d'exécution et la sensibilité aux interférences dans un cas artificiellement stressé au maximum. Ces informations permettent dans un second temps de sélectionner de façon pertinente les tâches qui constitueront un cas de test sur lequel implémenter le mécanisme de Surveillance et Contrôle. Le cas de test doit se composer d'une part d'un ensemble de tâches critiques d'une part et non critiques d'autre part. La caractérisation sert à identifier les tâches suivant des critères de stabilité et fiabilité (c.-à-d. variabilité d'exécution pour une même entrée), sensibilité aux interférences et part d'interférences générées par l'usage de ressources partagées.

Ce processus est relativement important pour quantifier les effets de notre mécanisme. De fait, la problématique d'interférences inter-tâches à laquelle nous essayons de répondre est hautement dépendante du comportement des tâches, notamment par leur usage des ressources partagées. Pour cette raison, il est important d'avoir

une bonne connaissance sur le logiciel sur lequel s'intègre un mécanisme censé limiter les conséquences de ces interférences. Cette démarche de caractérisation d'un jeu de tâches peut avoir son importance de la même manière pour un cas d'application industrielle déjà existant. De fait, particulièrement dans l'usage de logiciel en boîte noire, il est essentiel d'avoir une bonne maîtrise de ses caractéristiques d'exécution. Particulièrement dans la connaissance du taux d'utilisation de chaque ressource et de leur influence sur les temps d'exécution. Cela offre des éléments de décision pour l'allocation physique et d'ordonnancement des tâches. De cette façon le risque d'erreur d'estimation ou d'effets de bords dans les choix d'intégration peuvent être anticipés, voire limités.

Dans le cadre de nos expérimentations, l'objectif de ce protocole est de mesurer les métriques propres aux interférences inter-tâches : l'empreinte mémoire des tâches, le nombre de lectures/écritures, le nombre d'appels système, la puissance de calcul utilisée (taux CPU) et bien entendu les temps d'exécution et la sensibilité à l'application d'un stress sur les ressources partagées. De cette analyse, on se focalisera donc sur l'exploitation des tâches qui sont relativement sensibles à ces interférences d'une part, et à celles qui ont une forte empreinte mémoire et/ou une grande demande en ressources de calcul d'autre part. Cette démarche permet aussi pour un jeu de tâches préexistant, qui n'a pas été conçu pour un contexte de tâches temps-réel tel qu'on l'envisage, d'éliminer celles qui ont un comportement non maîtrisé qui ne correspond pas à un minimum que l'on serait en droit d'espérer de logiciel embarqué (par exemple une tâche qui aurait des variations de temps d'exécution sans interférences d'un ou plusieurs ordres de grandeur entre les millisecondes et la seconde).

TABLE 4.1 – Protocole de caractérisation d'un jeu de tâches

	Isolation pas d'interférences	Interférences artificielles	Interférences cas réel
Tâches Individuelles	1 Profil d'exécution de tâche	2 Profils de résilience aux interférences	N/A
Chaine de tâches	3 Profil d'exécution Chaine de tâches en isolation	4 <i>Profil de résilience Chaine de tâches</i>	5 Profil d'exécution cas réel

Pour arriver à cet objectif, nous allons travailler sur deux paramètres dans le cadre de ce protocole expérimental. D'une part nous allons faire varier la charge utile, qui est le logiciel monitoré pour en tirer ses caractéristiques, et d'autre part nous modifions les interférences – le stress – subit par la charge utile. Les différentes étapes du protocole sont agrégées dans le Tableau 4.1. La charge utile sera au choix soit une tâche spécifique, soit une chaîne de tâches telle que définie dans le chapitre précédent, tandis qu'elle sera soumise soit à :

- a) aucune interférence, donc en isolation,
- b) des interférences imposées de façon artificielles par le biais de logiciels dédiés

- (comme **stress-ng** sous Linux) pour stresser au maximum le système,
- c) les autres tâches du système, qui impliquent leurs propres interférences sur les ressources partagées.

Suivant ces paramètres, le protocole de caractérisation se compose de cinq étapes, d'abord en caractérisant les tâches de façon individuelles, ensuite en caractérisant la chaîne de tâches critiques et enfin en vérifiant la pertinence des choix des tâches critiques et non critiques et de leurs paramètres, au regard de l'ordonnançabilité et de la présence d'interférences logicielles :

- Profil d'exécution de chaque tâche
 - ① tâches individuelles exécutées en isolation
 - ② tâches individuelles exécutées avec stress imposé
- Profil d'exécution de la chaîne de tâches critiques
 - ③ chaîne de tâches en condition d'exécution en isolation
 - ④ chaîne de tâches exécutée avec le stress artificiel
- Set de tâche complet
 - ⑤ chaîne de tâches & tâches non-critiques qui constituent le cas de test

4.2.1 Profil des tâches

Cette phase est nécessaire de façon à déterminer le profil d'exécution des tâches qui sont à notre disposition pour former un cas de test. Bien entendu, dans l'éventualité où l'on disposerait d'ores et déjà d'un jeu de tâches accompagné de leurs spécifications, il n'est pas nécessaire de réaliser ces étapes de profilage des tâches dans l'objectif d'y réaliser une sélection de tâches critiques et non-critiques pour constituer notre cas de test. Il est possible de passer directement à l'étape ③ pour caractériser la chaîne de tâches critiques au regard du système et vérifier la pertinence d'implémenter un mécanisme de surveillance et contrôle selon les variations de temps de réponse sur les tâches critiques.

Ceci étant dit, ces étapes peuvent tout de même être intéressantes dans un cas industriel où le jeu de tâches est déjà déterminé et spécifié en avance. Comme mentionné plus haut, cela peut apporter une connaissance plus détaillée des tâches et donc aider dans l'anticipation de risques d'interférences et dans le choix de l'allocation des tâches pour le prévenir.

L'objectif de la phase de profilage est donc d'avoir les connaissances suffisantes sur les tâches en termes de temps d'exécution à la fois en cas optimal (aucune interférence) et au cas où la tâche subit un stress sur l'utilisation de ressources partagées. À cet effet, on peut mesurer des éléments subsidiaires tels que l'espace mémoire occupé, le nombre d'appels systèmes et le profil de lecture/calcul/écriture des tâches pour une analyse plus fine. À l'issu de ces tests, on doit être capable de spécifier quelles seront les tâches sélectionnées pour constituer le cas de test. Certaines tâches seront simplement éliminées à cause de caractéristiques compromettantes telles que des temps d'exécution moyens trop variables ou différents des autres tâches pour cohabiter ou des problèmes de non-déterminisme. D'autres pourront être éligibles en tant que tâches critiques, de part une bonne fiabilité d'exécution. Enfin et surtout, il

faudra inclure dans les tâches non-critiques au moins une part de tâches qui ont une forte influence sur l'utilisation des ressources partagées. Elles seront donc propices au déclenchement d'interférences sur les tâches critiques. Cette phase inclut donc les étapes ① et ②.

① - Tâches en Isolation Cette première phase du processus consiste à exécuter une tâche individuelle sur la plateforme d'intégration en lui fournissant des données d'entrée de façon périodique. Sont mesurées les informations susmentionnées pour un nombre représentatif d'exécutions au regard de la variété possible des données d'entrée.

Dans le cas où l'on ne connaît pas l'ordre de grandeur des temps d'exécutions, il faudra prendre dans un premier temps une période d'exécution arbitraire relativement large (toutes les secondes par exemple). Il est possible ensuite de réduire ce temps pour que les tests soient plus rapides (toutes les 50ms pour une tâche qui prend autour de 20ms typiquement). Si le jeu de tâches dispose d'ores-et-déjà de spécifications sur les temps d'exécutions moyens, il est possible de choisir directement une valeur proche avec une marge de sûreté pour être sûr de la complétion systématique de la tâche.

Ces mesures expérimentales doivent être reproduites pour chaque tâche, de façon à obtenir une table des caractéristiques de chaque tâche en isolation.

② - Tâches avec stress imposé artificiel De la même façon que pour l'étape précédente, l'objectif est d'obtenir les mêmes informations de profilage des tâches dans une mise en situation où le matériel est soumis à un niveau de stress élevé sur l'utilisation des caches, des entrées/sorties et en usage CPU, tandis que la tâche testée est exécutée périodiquement pour une durée suffisante de façon à obtenir un échantillon représentatif. Cela est dépendant du système et de la tâche en question.

Les étapes avec un stress imposé de façon artificielle sur le matériel par l'usage d'outils dédiés au stress. Plusieurs solutions sont possibles selon la plateforme de développement impliquée (matériel et logiciel). Pour mentionner quelques exemples, dans le cadre des travaux de [Blin 2016a] des tâches dédiées ont été codées en assembleur pour effectuer des cycles d'accès mémoire sur une plateforme multi-coeur SABRE Lite. Sur des systèmes basés sur UNIX, il existe des outils comme **stress-ng** [King 2019] qui offrent toute une librairie d'utilitaires pour stresser des éléments spécifiques du processeur.

À l'issue de ces deux étapes de caractérisation, on dispose d'une matrice des caractéristiques du jeu de tâches qui inclut, pour chaque tâche telle qu'illustré dans le Tableau 4.2. On y trouve les temps d'exécution minimum, médian, moyen et maximum, le nombre d'appels système, et éventuellement d'autres métriques propres à l'environnement d'exécution si elles sont pertinentes (nombre d'appels à un composant spécifique du firmware par exemple).

À partir de ces données, on peut alors identifier :

1. Les tâches qui ne sont pas adaptées à nos besoins pour un cas de test. Soit parce que leurs temps d'exécution ne sont pas à la bonne échelle, c'est-à-dire avec des écarts d'ordre de grandeur trop importants. Par exemple des tâches

trop macroscopiques qui prennent plusieurs secondes de calcul alors que le reste du système de tâche est composé de micro-tâches monofonctionnelles qui s'exécutent en quelques dizaines de millisecondes. Soit parce qu'elles ont été codées d'une manière qui ne semble pas acceptable, par exemple avec une surutilisation des appels système, qui provoquent des ralentissements indésirables trop dépendants du système d'exploitation.

2. Les tâches qui sont appropriées pour servir de tâches non critiques qui vont provoquer des interférences. Ces dernières peuvent avoir des variations de temps d'exécution non négligeables et utiliser des ressources partagées.
3. Enfin, les tâches qui peuvent servir en tant que tâches critiques, qui représentent alors à minima des tâches temps-réel souple avec peu, voire pas d'appels système et un fonctionnement maîtrisé en isolation (variations de temps d'exécution raisonnables), et potentiellement sensibles à des interférences, donc avec une variation non négligeable du temps d'exécution médian entre les 2 conditions de test.

TABLE 4.2 – Template type des données de caractérisation des tâches

Tâche	Appels Sys.	Autres appels	en isolation			stressé		
			t_{min}	t_{max}	t_{med}	t_{min}	t_{max}	t_{med}
τ_1	—	—	—	—	—	—	—	—
...	—	—	—	—	—	—	—	—
τ_n	—	—	—	—	—	—	—	—

À l'issu de cette sélection, on a donc à disposition un jeu de tâches, qui va être constitué d'une liste de **tâches non critiques** avec une période d'exécution pour chacune d'entre elle, et une liste de **tâches critiques** qui constituent la chaîne de tâches, qui doit donc inclure toutes les données relatives à la chaîne de tâches : période d'exécution de chaque tâche, mais aussi contraintes de précédence et échéance de temps de réponse bout-en-bout.

Il s'avère que cette dernière donnée va être hautement dépendante des spécifications système dans le cadre d'un cas applicatif réel.

Dans une situation purement expérimentale où il nous faut constituer nous-même le cas de test, cette donnée de temps de réponse bout-en-bout doit être fixée arbitrairement. Par conséquent, il va falloir là encore proposer une façon de fixer cette contrainte, ou au moins en définir un ordre de grandeur qui pourra être ajusté selon la difficulté que l'on souhaite imposer au test. Plus cette échéance est courte, plus il sera difficile de la garantir au regard des pires temps d'exécution des tâches et des risques d'allongement d'exécution inhérents notamment aux interférences.

4.2.2 Profil du cas de test et Chaîne de tâches

L'objectif de cette seconde phase de caractérisation consiste justement à caractériser plus globalement les tâches sélectionnées, notamment par la vérification de

l'ordonnançabilité et de la contrainte de temps de réponse bout-en-bout de la chaîne. Il s'agit des étapes ③, ④ et ⑤ du protocole.

③ - Chaîne de tâches critique en isolation Il est important de tester l'exécution de la chaîne de tâches en isolation, c'est-à-dire sans aucune interférence provenant des autres tâches qui doivent être intégrées. Cette étape permet de vérifier l'absence d'effets de bord entre la situation où les tâches étaient testées individuellement et le cas où la chaîne de tâches est exécutée sans aucune autre tâche non critique, donc en isolation. Le mode que l'on souhaite caractériser ici correspond, dans le cadre de l'utilisation du mécanisme de surveillance et de contrôle, au mode dégradé de fonctionnement. C'est-à-dire le mode d'exécution de la chaîne de tâches avec réduction des interférences par mise en pause des tâches non critiques. Cette caractérisation permet de définir le temps de réponse bout-en-bout que l'on sera capable de garantir en toute circonstance grâce à un passage en mode dégradé. En exécutant directement les tâches critiques en isolation, on peut connaître le profil de temps de réponse de la chaîne de tâches : temps de réponse moyen et pire temps de réponse bout-en-bout. À partir de cette information, il devient possible de fixer dans notre spécification l'échéance de temps de réponse maximale permise pour la chaîne. Plus cette échéance sera proche du temps de réponse en isolation, plus cela implique que cette échéance est stricte dans le sens où elle est plus courte et donc difficile à tenir en cas de perturbations.

Dans le cadre de nos expérimentations, nous fixons à cette étape deux éléments pour avancer dans les tests. Le premier sur la cohérence de la chaîne de tâches. Les tâches impliquées ici ayant un certain niveau de criticité, il semble pertinent de partir du principe qu'en l'absence de perturbations significatives l'écart entre le temps d'exécution minimal et maximal constatés pour la chaîne de tâches doivent être relativement restreints. Par conséquent, il faudra réviser le choix des tâches sélectionnées pour la chaîne dans le cas où l'on observe un écart trop significatif entre $WCRT_{max}$ et $WCRT_{min}$. Typiquement, si cet écart est supérieur la moitié du temps de réponse médian observé. Ce qu'on l'on peut formaliser comme : si $WCRT_{max} - WCRT_{min} \leq 0.5 * WCRT_{median}$ en isolation, alors la chaîne est suffisamment cohérente dans son comportement nominal. Si cette conclusion est remplie, alors le second élément sera l'échéance de temps de réponse de référence que l'on pourra se fixer comme spécification, qui sera $D_C = 2 * WCRT_{median}$. Il s'agit d'un choix réaliste qui peut être revu en fonction d'un contexte de système critique spécifique. Cela implique d'abord qu'en isolation, par définition on est garanti que la chaîne de tâches critiques est ordonnable et ne peut pas dépasser son échéance et ensuite que l'on pourra ajuster cette échéance pour tester l'efficacité du mécanisme de Surveillance et Contrôle, en ayant cette valeur de référence comme base pour tester une spécification plus contraignante et donc difficile à respecter en cas d'interférences avec une échéance plus courte ou inversement .

④ - Chaîne de tâches sous stress imposé Il est possible de tirer parti des tâches de stress qui ont pu être utilisées pour la caractérisation individuelle de l'étape ② pour caractériser de la même façon la chaîne de tâches critiques qui a été sélectionnée. L'objectif est de s'assurer de la pertinence d'implémenter le

mécanisme de Surveillance et Contrôle dans le cas d'étude que nous sommes en train de construire, en vérifiant que la chaîne de tâches peut effectivement être sujette à des interférences qui mènent à un dépassement d'échéance. Le test consiste logiquement à exécuter la chaîne de tâches critiques sélectionnée avec les paramètres de l'étape précédente, conjointement aux tâches dédiées à la mise en stress du processeur sur les ressources partagées. En toute logique, on doit constater des dépassements de l'échéance de manière significative. Si aucune variation n'est constatée comparé au cas où la chaîne est isolée, alors les tâches sélectionnées ne sont pas sujettes aux interférences et donc il ne sera pas utile d'y adjoindre de mécanisme de contrôle. La chaîne choisie n'est a priori pas pertinente pour un cas de test.

⑤ - Système complet dont chaîne de tâches critiques Dans le cadre de mise en application de systèmes temps-réel à criticité mixte, il est nécessaire de passer par une phase de vérification d'ordonnançabilité. Cette phase est déjà partie intégrante des processus de développement industriels et propose de nombreux outils d'analyse tels que cela a été décrit dans le chapitre 2. Nous n'avons pas approfondi en détail cette dimension de l'implémentation d'un set de tâches pour nous focaliser sur la proposition d'un mécanisme en lui-même en supposant que les vérifications d'usage d'ordonnançabilité sont validées. Il faut notamment noter que notre démarche véhicule l'espoir d'alléger les contraintes de garanties d'ordonnançabilité dans une certaine mesure. Notre positionnement étant que par la possibilité de passer en cas d'isolation qui permet de façon transitoire la garantie d'ordonnançabilité d'un sous-ensemble des tâches alors les preuves d'ordonnançabilité par estimation de pire temps d'exécution peuvent être allégées sur les tâches ciblées par le mécanisme.

Par conséquent, dans cette étape, nous réalisons les vérifications d'ordonnançabilité du jeu de tâches sélectionnées sans rentrer dans des considérations trop approfondies tel que de la vérification formelle. De fait, on peut se permettre des dépassements d'échéances de façon marginale sur les tâches non-critiques (par définition) et sur les tâches critiques (avec la perspective de compenser cela via le mécanisme). Par conséquent, cette vérification peut se faire avec un niveau d'exigence et de certitude du résultat moins strict. Une possibilité consiste à vérifier d'abord de façon grossière via un calcul de taux d'occupation CPU moyen qui peut être ajusté avec la période d'exécution des tâches suivant la formule : $\sum C_i/T_i \leq 1$ pour toutes les tâches allouées au même cœur pour un ordonnancement partitionné. Dans le cadre d'un ordonnancement global sur un processeur à N coeurs, on a aussi $\sum C_i/T_i \leq N$ (c.-à-d. avec possibilité de migration des tâches entre les coeurs). Et dans un second temps par une vérification expérimentale avec le cas de test complet tel que défini précédemment pour constater l'absence de dépassement d'échéances sauf éventuellement de façon marginale. Il existe des méthodes analytiques plus précises pour déterminer l'ordonnancabilité d'un set de tâches sur un calculateur multicœur. On pourra notamment mentionner [Melani 2016] qui semble fort approprié dans le cadre de notre étude orientée sur des chaînes de tâches, ou encore plus généralement [Chen 2014] ou [Han 2018] pour des ordonnancements partitionnés. Si nous n'utilisons pas de telles méthodes, c'est avant tout parce qu'elles sont plus coûteuses en temps et complexité de mise en place. Elles peuvent nécessiter aussi des connaissances plus fines pour modéliser le système. Cela peut être parfaitement

approprié, mais pas nécessaire dans notre cadre, d'autant plus pour un contexte industriel pour lequel des estimations approximatives est souvent suffisant.

Dans un second temps la vérification est faite expérimentalement en surveillant l'exécution correcte des tâches selon les périodes fixées dans une mise en situation réelle la plus représentative possible. La question des conditions expérimentales est très riche et pourrait constituer un sujet de recherche en lui-même. Il est notamment question de définir la durée ainsi que la richesse des conditions de tests (représentativité des données d'entrée du test). En effet, dans l'idéal, il faut s'assurer que les durées d'expérimentation ainsi que la diversité des conditions de tests permettent une couverture suffisamment exhaustive pour couvrir tous les cas de fonctionnement du logiciel, même les plus spécifiques. Dans le cas où l'on a à disposition les codes sources, cela peut être facilité par modélisation et étude analytique de façon à générer un modèle de test qui couvre tous les cas. Mais cela n'est plus possible à partir du moment où au moins une partie du logiciel est en boîte noire ou bien tout simplement si le logiciel a un niveau de complexité qui rend impossible de couvrir de façon exhaustive toute la combinatoire des possibilités d'exécution. Dans ce cas présent, nous sommes tenus à faire un choix d'un échantillon de test, classiquement de façon aléatoire. De même pour la durée de test, il faut *a minima* considérer une durée qui permette à la fois d'avoir un régime de fonctionnement stable sur les mesures et une durée suffisante pour avoir un nombre de mesures qui limite statistiquement les marges d'erreur. Nous faisons donc le choix de données d'entrée aléatoire pour les tâches, avec une durée arbitraire de l'ordre de la minute qui nous a paru suffisant pour constater une stabilité des mesures. Il serait bien entendu possible d'affiner et perfectionner ces choix, mais ils présentent d'ores-et-déjà une base sur laquelle on pourra discuter et obtenir des premiers résultats.

Cette étape permet aussi d'ores-et-déjà de constater le comportement de la chaîne de tâches critiques dans le cas nominal en l'absence de mécanisme pour contrôler d'éventuels augmentations indésirables de son temps de réponse. Il faut comparer ces temps de réponse avec les résultats obtenus à l'étape ③, qui correspond au cas de la chaîne est en isolation. Cela permet de quantifier l'influence spécifique à notre cas d'étude des interférences sur les tâches critiques et d'étudier la pertinence du choix d'échéance bout-en-bout qui a été fait à cette même étape. De fait, selon les temps de réponse constatés en fonctionnement normal sans mécanisme, sachant que la vérification de l'influence d'interférences a été effectuée à l'étape précédente, il est alors possible de voir si les tâches non-critiques choisies ont effectivement, elles aussi, un effet significatif sur l'usage des ressources partagées. Si la chaîne de tâches critiques conserve un profil d'exécution similaire au cas en isolation, on est en mesure de dire que les tâches non-critiques choisies ne sont pas suffisantes pour avoir une influence sur les ressources partagées.

Après avoir validé la définition des tâches critiques à l'étape précédente, on valide dans cette étape finale la définition des tâches non-critiques et donc du cas de test dans son ensemble.

En conclusion, ce protocole simple en 5 étapes permet à partir d'un jeu de tâches dont nous ne connaissons pas les caractéristiques de spécifier un jeu de tâches contenant d'une part une chaîne de tâches critiques et d'autre part des tâches non critiques, avec leurs périodes, échéances implicites et répartition sur les coeurs.

Ces étapes permettent par la même occasion de s'assurer un choix pertinent de tâches où le problème de respect d'échéance se pose au regard de l'influence des interférences sur les tâches critiques qui provoquent des retards d'exécutions qui peuvent potentiellement provoquer une faute temporelle.

4.3 Protocole d'Implémentation et Calibration

En disposant d'un cas d'étude avec ses spécifications habituelles d'intégration du logiciel sur un calculateur multicoeur, il nous est possible d'appliquer ce protocole de façon à calibrer le Mécanisme de Surveillance et Contrôle afin de garantir le respect d'échéance d'une chaîne de tâches critiques spécifique. Ce qui était désigné comme le cas en isolation dans le protocole de spécification précédent correspond alors, dans le cas de l'usage du mécanisme de Surveillance et de Contrôle, au passage en mode dégradé qui désactive les tâches non critiques pour retirer les interférences.

De la même façon que pour le protocole de Conception d'un Cas d'Étude, nous nous basons ici aussi sur la mise au test de la chaîne dans 3 contextes différents : en mode dégradé, avec des interférences matérielles imposées de façon artificielle et enfin avec toutes les tâches non-critiques, en d'autres termes le système complet. Le protocole se divise en deux temps avec :

1. Dans un premier temps la configuration des paramètres du mécanisme de Surveillance et de Contrôle. Il est alors exécuté uniquement pour sa composante de Surveillance pour récupérer les métriques nécessaires propres au système (temps de passage en mode dégradé, caractérisation des rWCRT de la chaîne...).
2. Dans un second temps, la composante de Contrôle est aussi activée pour tester effectivement le mécanisme selon les paramètres configurés, en mesurer les différentes métriques d'efficacité, et potentiellement réitérer sur le réglage des paramètres pour obtenir le comportement désiré.

En résumé, et comme indiqué dans la Tableau 4.3, pour calibrer le mécanisme de Surveillance et Contrôle implémenté, il est nécessaire de passer par 5 étapes :

- **Spécification des paramètres** – la chaîne de tâches est exécutée uniquement avec les fonctionnalités de Surveillance
 - ① Système exécuté en mode dégradé – uniquement la chaîne de tâches
 - ② Système complet exécuté – tâches non-critiques incluses
- **Validation et mesures de comportement** – la chaîne est exécutée avec le mécanisme complet activé
 - ③ Chaîne de tâches en conditions d'exécution du mode dégradé
 - ④ Chaîne de tâches exécutée avec charge de stress artificiel
 - ⑤ Système complet exécuté – tâches non-critiques incluses et mécanisme actif

TABLE 4.3 – Protocole d'implémentation et calibration du mécanisme

	Mode Dégradé	Interférences artificielles	Interférences cas réel
Chaine de Tâches + Surveillance	1 Caractérisation Chaine de tâches en mode dégradé	N/A	2 Profil d'exécution cas réel
Chaine de Tâches + Surveillance + Contrôle	3 Vérification Sensibilité	4 Vérification Fiabilité	5 Profil d'exécution cas réel Efficacité et Calibrage

4.3.1 Spécification des paramètres de Contrôle

L'étape ① de la calibration du mécanisme consiste à déterminer les grandeurs de paramétrage de ce dernier. Il est nécessaire de définir :

- t_{sw} le temps de passage en mode dégradé ;
- T_{CCC} la période d'exécution du mécanisme de Surveillance et Contrôle ;
- $rWCRT(\tau_p)$ pour chaque tâche τ_p de la chaîne.

Le temps de passage en mode dégradé se calcule simplement de façon expérimentale, par mesures successives de déclenchement forcé du mode dégradé. Une valeur de borne maximale des mesures obtenue permet de garantir une estimation sûre de cette valeur. Dans les faits, cette valeur est relativement faible, voire négligeable. En effet, le passage en mode dégradé correspond à l'envoi d'un signal système qui met en pause les tâches non critiques. La durée de cette action dépend donc de deux éléments, d'abord du temps de réception du signal et ensuite du temps de mise en pause de la tâche. Ces derniers sont relativement faibles et la seule source de délai repose sur le temps de sauvegarde de contexte pour les tâches non critiques qui sont actuellement en cours d'exécution lors de la réception du signal.

① – Système exécuté en mode dégradé Cette étape a pour objectif de déterminer deux des grandeurs susmentionnées. T_{CCC} la période de vérification du mécanisme de contrôle. Et les différents $rWCRT(\tau_p)$ qui indiquent chacun la pire temps d'exécution restant garanti en mode dégradé pour la chaîne de tâche, selon le nombre de tâches restantes à exécuter pour la terminaison de la chaîne.

La chaîne de tâches est exécutée dans les mêmes conditions que le mode dégradé du système, avec le mécanisme qui est exécuté uniquement pour sa composante de Surveillance. Il convient donc de fixer en premier lieu le choix de T_{CCC} , qui est déterminé pour permettre au mécanisme implanté de gérer la charge de réception des timestamps de terminaison des tâches critiques d'une part, et par le niveau de sensibilité à l'anticipation que l'on souhaite donner au mécanisme d'autre part. Il est ainsi hautement dépendant des spécifications de la partie critique du système. Pour donner une première valeur de période, nous nous focalisons sur la charge de réception. Il faut à chaque instant garantir qu'aucun timestamp ne soit perdu.

Par conséquent, soit L_{BUF} le nombre d'emplacements disponibles dans la mémoire tampon de réception des timestamps et n le nombre de tâches critiques ayant pour périodes d'exécution $T_i, i \in \llbracket 1, n \rrbracket$, alors la condition suffisante pour garantir le traitement de tous les timestamps est :

$$L_{BUF} \geq n \quad \text{et} \quad T_{CCC} \leq \min(T_i) \quad (4.1)$$

De cette façon, il est garanti qu'à chaque instant t , en pire cas le Core Control Component aura au maximum n timestamps à traiter, dans l'hypothèse –peu probable– où toutes les tâches critiques se terminent en même temps. De plus la récupération des timestamps doit être plus fréquente que l'occurrence de toute tâche critique, en particulier celle dont la période est la plus petite. En d'autres termes, le flux de traitement des timestamps par le mécanisme doit être supérieur au flux de terminaison des tâches critiques. Avec ces deux conditions, on a la garantie de ne perdre aucun timestamp envoyé. De fait, en considérant que chaque emplacement de la mémoire tampon correspond à une des tâches critiques, on a bien au moins 1 emplacement pour chaque tâche (première inéquation). Donc avec le mécanisme de contrôle qui s'exécute plus souvent que chacune des tâches critiques, on est sûr qu'il n'y aura pas plus d'un timestamp par tâche critique à gérer à chaque période de contrôle. On est donc sûr que la mémoire tampon n'aura pas de débordement et donc qu'aucun timestamp ne sera perdu.

Ces deux conditions de l'Équation 4.1 sont suffisantes, mais pas nécessaires. Cela fournit un premier couple de valeurs de période d'exécution et de taille de mémoire tampon simples et fonctionnels. Dans un cadre où la contrainte de mémoire disponible est très contraignante, il est probable que selon la chaîne de tâches étudiée, on puisse réduire taille de mémoire tampon, en calculant le nombre maximal de terminaison de tâches critiques possible durant une période de temps T_{CCC} . Ou inversement, pour diminuer l'empreinte mémoire du mécanisme de surveillance et de contrôle, cette valeur de T_{CCC} pourra être affinée en étape ⑤. Cela permet par la même occasion de jouer sur le niveau de sensibilité à l'anticipation du mécanisme, si nécessaire.

Le mécanisme de surveillance est activé pour conserver les traces d'exécution de la chaîne de tâches $ET(j, t)$. Pour rappel, chaque trace d'exécution enregistre le timestamps de fin d'exécution de toutes les tâches de la chaîne selon le respect de la contrainte de précédence dans la chaîne. On reconstitue donc à partir de ces traces les temps d'exécution intermédiaires de la chaîne de tâche :

pour chaque $p \in \llbracket 1; n - 1 \rrbracket$, $ET(j, t = succ^p(e_{1,j})) = succ^p(e_{1,j}) - a_{1,j}$.

Comme on a pu le décrire en chapitre 3, les $e_{1,j}$ et $a_{1,j}$ correspondent respectivement aux dates de fin et d'activation du $j^{\text{ème}}$ job de la première tâche critique de la chaîne. Et $succ^p(e_{1,j})$ le timestamp de terminaison du $p^{\text{ème}}$ successeur du job d'entrée $\tau_{1,j}$ de la chaîne de tâches.

En compilant tous les temps d'exécution intermédiaires obtenus, on peut en tirer des bornes maximales, qui correspondent alors aux différents $rWCRT(\tau_p)$, on obtient :

$\forall p \in \llbracket 1 ; n - 1 \rrbracket$, toutes les traces d'exécution obtenues :

$$rWCRT(\tau_p) = \max_{j \in \mathbb{N}^*} (ET(j, succ^p(e_{1,j}))) = \max_{j \in \mathbb{N}^*} (succ^p(e_{1,j}) - a_{1,j}) \quad (4.2)$$

Les $rWCRT(\tau_p)$ alimenteront donc la partie Contrôle du mécanisme pour son bon fonctionnement.

② – Système complet avec Surveillance Cette étape permet d'obtenir deux informations. La première essentielle est la valeur de t_{SW} , le temps de passage en mode dégradé. Et la seconde étant la vérification de la pertinence d'utilisation du mécanisme de Surveillance et Contrôle.

De fait, en exécutant le système en condition normale de fonctionnement, adjoint au mécanisme de Surveillance sans activer le Core Control Component, il est possible de déclencher des transitions forcées aléatoires vers le mode dégradé, et d'en mesurer le temps de transition. Une borne supérieure aux valeurs mesurées permet de fixer t_{SW} .

Par ailleurs, à l'image de ce qui a été fait dans le protocole de spécification d'un cas de test, en section 4.2.2, on peut tirer profit de cette étape pour constater les potentielles fautes temporelles par dépassement de l'échéance bout-en-bout de la chaîne de tâches grâce à l'outillage de mesure permis par la Surveillance. Cette information de taux de dépassement d'échéance sur une mise en pratique expérimentale représentative offre une bonne base comparative pour les résultats ultérieurs qui seront obtenus après activation du mécanisme de contrôle.

Suite à ces deux étapes, nous avons à disposition toutes les données et paramètres nécessaires à la mise en place du mécanisme complet de Surveillance et de Contrôle. L'objectif des étapes qui suivent dans le protocole consiste donc à quantifier l'efficacité du mécanisme. On s'en servira notamment comme information de retour pour quantifier l'influence d'une modification sur les paramètres précédemment fixés.

4.3.2 Calibration du Mécanisme de Contrôle

③ – Sensibilité du mécanisme Un premier test réalisable est d'exécuter à nouveau le système directement en mode dégradé, avec les tâches non critiques qui ne sont pas exécutées, mais le mécanisme complètement actif. De cette façon, on mesure la sensibilité du mécanisme par l'étude du comportement du mécanisme dans un cadre dans lequel *a priori* il ne devrait pas se déclencher de par l'absence d'interférences qui pourraient causer une faute temporelle. En récupérant le nombre de passages en mode dégradé dans de telles conditions expérimentales, on peut donc identifier s'il y a une piste d'amélioration des paramètres du mécanisme, notamment sur sa période d'exécution. De fait, plus on observerait un nombre élevé de changement de modes, plus cela signifie que le mécanisme est trop sensible. Cela peut être alors dû à deux éléments :

- Soit la période d'exécution du mécanisme de contrôle est trop grande, ce qui fait que la constante W_{MAX} dans Équation 3.1 de l'anticipation est trop prépondérante. Cela provoque une anticipation qui se base sur un point de contrôle suivant trop lointain. En d'autres termes, l'anticipation vérifie qu'il

n'y a aucun risque de dépassement d'ici au prochain point de contrôle et plus le prochain point de contrôle est distant moins le mécanisme ne pourra accepter d'écart dans les temps d'exécution. Pour obtenir une bonne anticipation du risque de viol de l'échéance la prise de décision devrait se baser en premier lieu sur les grandeurs dynamiques du système, qui évoluent au fil de l'exécution, donc $RT(t)$ et $rWCRT(t)$.

- Soit les ordres de grandeur impliqués entre les temps d'exécution des tâches avec et sans perturbations sont trop proches au regard des autres grandeurs du système (période de surveillance notamment), ce qui rend la différenciation entre un cas à risque et un cas nominal bien trop subtil pour être discriminé par le mécanisme. Cela peut être dû aux estimations trop pessimistes qui ont été faites sur les pires temps de réponse restants, autrement, il est possible que tout simplement l'échéance fixée est trop stricte et en conséquence trop proche des temps de réponse constatés lors d'un fonctionnement nominal sans aucune interférence. Auquel cas ce genre de mécanisme n'est pas adapté et il faudra se contenter des méthodes plus classiques de contrôle statique au détriment de l'optimisation de la puissance de calcul.

Dans l'idéal, il faudrait qu'aucun déclenchement de mode dégradé ne soit déclenché dans ce cadre-ci, auquel cas on sait même que l'on pourrait se permettre potentiellement une plus grande période d'exécution du mécanisme pour alléger son empreinte sur l'utilisation du processeur. À condition bien entendu de rester dans les conditions pour pouvoir gérer la réception de tous les timestamps bien entendu.

④ – Fiabilité du Mécanisme En exécutant la chaîne de tâches critique avec le Contrôle opérationnel dans un contexte où le processeur est le plus stressé possible en terme d'interférence, il est possible de mesurer la fiabilité du mécanisme. Dans un tel contexte pire cas, il est possible de vérifier si les choix des paramètres de contrôle ont été correctement choisis. Si tel est le cas, on ne devrait observer aucun dépassement d'échéance lors de ces tests. S'il advenait que l'échéance de temps de réponse bout-en-bout n'est pas respectée un certain nombre de fois non nul, alors il est essentiel d'approfondir la question pour remonter aux raisons de ces fautes.

Il peut y avoir ici deux explications à de tels dépassements. Il peut s'agir simplement d'une sous estimation des $rWCRT(\tau_p)$, qui a mené à un déclenchement du mode dégradé trop tardif. Cela doit pouvoir s'observer en observant la trace d'exécution de la chaîne de tâches qui a conduit au dépassement. On devrait observer un profil d'exécution dans les tranches hautes de temps de réponse habituellement observés pour la chaîne de tâches (et donc proche, mais inférieur aux valeurs de $rWCRT(\tau_p)$ paramétrées), mais pas suffisamment pour que le déclenchement du mode dégradé se soit fait assez tôt.

En revanche en cas de comportement avec un profil de temps de réponse en cours d'exécution plutôt habituel, pour lesquels le mécanisme de Contrôle semble s'être déclenché dans les temps, mais le temps d'exécution bout-en-bout a continué à croître jusqu'à atteindre le dépassement, alors il est possible que ce dépassement soit dû à des facteurs non contrôlés par le mécanisme et donc plus spécifiquement des retards d'exécution qui ne sont pas dû à des interférences matérielles. Si ce cas

est souligné ici, c'est parce qu'il est vraisemblable lorsque ce genre de mécanisme est utilisé dans un environnement complexe où tout ce qui se produit au niveau du système d'exploitation n'est pas parfaitement maîtrisé. Cela est particulièrement le cas avec l'usage de plus en plus fréquent de distributions Linux pour les systèmes embarqués. Plusieurs travaux tels que [Allende 2019] ou encore [Sivakumar 2020] et [Serra 2020] se sont intéressés aux conditions de mise en place d'un tel système pour du temps-réel, les difficultés que cela implique ainsi que ce que l'on peut attendre de Linux dans ces conditions. Cette étape de vérification de la fiabilité peut donc permettre de soulever des problèmes externes propres au matériel ou au logiciel sous-jacent, qui devront être résolus par ailleurs.

⑤ – Système complet avec mécanisme de contrôle Pour finir, la dernière étape du protocole de calibration et implémentation du mécanisme consiste simplement à exécuter la totalité du système dans son fonctionnement nominal, accompagné du mécanisme de surveillance et de contrôle. Cette dernière phase d'essai a, elle aussi, pour objectif d'apporter des métriques de mesure de qualité de fonctionnement du mécanisme. Nous avons mesuré jusqu'à présent la sensibilité et la fiabilité du mécanisme. Il reste alors l'efficacité, qui demande de regarder dans quelle mesure nous réussissons à conserver l'exécution des tâches non critiques malgré l'ajout du contrôle pour garantir le respect de l'échéance critique bout-en-bout. En réalisant ce test dans les mêmes conditions que l'étape ② où la partie Contrôle était inactive, on peut donc comparer le temps d'exécution et le nombre de jobs réalisés pour chaque tâche non critique, au regard de la diminution du nombre de dépassements d'échéances. On devrait normalement constater un taux de dépassement de l'échéance qui devient nul en cas d'une fiabilité de 100%. Par la même occasion, on peut mesurer le temps d'exécution moyen du mécanisme de surveillance et contrôle en fonctionnement nominal, de façon à vérifier que son empreinte d'utilisation du CPU est non significative et n'empiète donc pas sur les performances globales, ce qui serait bien entendu contre-productif.

Cette étape est enfin l'occasion de faire varier les paramètres de configuration du Contrôle en réitérant sur les 2 étapes précédentes, pour ajuster les performances mesurées du mécanisme aux besoins spécifiques de l'application si nécessaire. Ainsi il sera possible de modifier la période d'exécution du Contrôle soit pour diminuer son utilisation du CPU dans le cas d'un taux d'utilisation considéré trop élevé, soit parce que l'on a constaté une sensibilité trop élevée, par un nombre de faux positifs trop grand. Il est aussi possible d'accepter de façon exceptionnelle des dépassements de l'échéance bout-en-bout, dans un modèle où la chaîne serait à temps-réel souple typiquement ou si l'on veut garantir un niveau de Qualité de Service (*QoS*) comme cela a pu être développé dans des études comme [Lin 2006]. Pour ce faire, on peut revoir les valeurs de $rWCRT(\tau_p)$ à la baisse. Cela revient à fixer les pires temps d'exécution restant garantis en mode dégradé à des valeurs plus optimistes. Par conséquent, la sensibilité du mécanisme sera allégée au détriment de dépassements d'échéance de façon occasionnelle malgré le passage en mode dégradé, typiquement quand le temps de réponse en mode dégradé sera situé juste au dessus des nouveaux seuils de $rWCRT(\tau_p)$ qui ont été abaissés. Cela donnera alors une fiabilité inférieure à 100% de garantie de l'échéance, mais qui peut être ajustée expérimentalement pour

rester à un taux d'occurrences maîtrisé. Il sera alors possible par calcul probabiliste de fixer une baisse de ces valeurs pour obtenir au niveau souhaité de taux de défaillance temporelle sur la chaîne de tâches critiques, typiquement.

Ces ajustements des paramètres du mécanisme sont des possibilités offertes par ce dernier, hautement dépendantes de besoins spécifiques. Nous étudierons plus en détail l'influence de variation de ces valeurs dans la suite des travaux. Le protocole expérimental d'analyse du mécanisme étant clos ici.

4.4 Protocole expérimental global

Pour la suite de nos travaux, comme l'on peut s'en douter nous avons dû mettre en œuvre les deux protocoles expérimentaux présentés ici. En premier lieu le protocole de spécification d'un cas de test mis en application sur une suite logicielle de benchmarking. Et ensuite le protocole d'implémentation du mécanisme de surveillance et de contrôle sur le cas de test précédemment spécifié. Par conséquent, nous présentons en conclusion de cette partie le tableau récapitulatif expérimental, dans le cas où ces 2 méthodes doivent être employées successivement.

Il est à noter que les deux protocoles se recoupent en partie, dans les conditions expérimentales de test de la chaîne de tâches critiques. Il s'agit de la seconde ligne du Tableau 4.1 où la chaîne de tâches est caractérisée, qui propose les mêmes conditions expérimentales que la première ligne du Tableau 4.3. La nuance à noter est que là où le protocole de Spécification propose une caractérisation de la chaîne de tâches *en isolation*, dans le cadre de l'implémentation du mécanisme de contrôle, ces conditions de test doivent correspondre au cas exact dans lequel sera le système en mode dégradé. Il s'agit alors d'un cas où la chaîne de tâche est isolée d'interférences des tâches non critiques, mais cela implique aussi une configuration spécifique des tâches critiques, notamment sur leur allocation physique sur les coeurs du processeur. En bref, tout cela pour exprimer que la caractérisation de la chaîne de tâche doit se faire dans les mêmes conditions que celles qui seront impliquées lors d'un passage en mode dégradé par le mécanisme de contrôle.

TABLE 4.4 – Protocole "Complet" de Spécification, Caractérisation et Analyse

	Mode Dégradé	Interférences artificielles	Interférences cas réel	
Tâches Individuelles	1 Profil d'exécution de tâche	2 Profils de résilience aux interférences	N/A	 Phase de Conception
Chaine de Tâches + Surveillance	3 Caractérisation Chaine de tâches en mode dégradé	4 <i>Profil de résilience Chaine de tâches</i>	5 Profil d'exécution cas réel	 Phase de Paramétrage
Chaine de Tâches + Surveillance + Contrôle	6 Vérification Sensibilité	7 Vérification Fiabilité	8 Profil d'exécution cas réel Efficacité et Analyse	 Phase d'Analyse

En bilan, nous avons donc le Tableau 4.4 qui donne 3 Phases expérimentales : la Conception, le Paramétrage et l'Analyse.

Lors de la phase de Conception, sont testées successivement les tâches de façon individuelle pour en obtenir une caractérisation temporelle. Cela permet d'avoir une meilleure connaissance des temps d'exécutions des tâches et de leur propension à introduire et/ou recevoir des interférences sur l'utilisation de ressources partagées. Dans le cas où il ne s'agit pas d'un jeu de tâches déjà spécifié pour un système industriel, il s'agit aussi de constituer la sélection d'un jeu de tâches de test à partir de tâches provenant d'un benchmark ou autre. Cela permet de spécifier les tâches non critiques et les tâches critiques sous forme d'une chaîne de tâches. Dans cette éventualité, la phase de conception propose enfin la vérification des choix effectués notamment en termes de taux d'utilisation du processeur et de la présence manifeste d'interférences qui peuvent mener à des dépassements d'échéances.

Vient ensuite la phase de Paramétrage, qui repose sur l'exécution de la chaîne de tâches critiques uniquement pour déterminer les paramètres du mécanisme de Surveillance et Contrôle.

Pour finir, la phase d'Analyse intègre la chaîne de tâches avec l'ajout du mécanisme complet. Chaque test est soumis soit à aucune interférence matérielle spécifique (cas en mode dégradé), soit avec des interférences imposées de façon artificielle pour s'approcher d'un "pire cas" de charge, soit avec la charge réelle qui inclut les interférences inhérentes à toutes les tâches du système, notamment les tâches non critiques. De ces tests on est en mesure de définir les propriétés du mécanisme de Surveillance et Contrôle selon des critères de Qualité, Performance et Fiabilité tel que présentés précédemment.

CHAPITRE 5

Implémentation, Résultats et Analyse

Sommaire

5.1	Plateforme de développement	80
5.1.1	Plateforme Matérielle	80
5.1.2	Support Logiciel	82
5.2	Logiciel Applicatif	86
5.2.1	Benchmark MiBench	86
5.2.2	Charge de test	88
5.2.3	Implémentation de la plateforme expérimentale logicielle . . .	89
5.3	Application du Protocole Expérimental à MiBench	93
5.3.1	Phase de Conception	94
5.3.2	Phase de Configuration du mécanisme	97
5.3.3	Phase de Validation et Analyses	102

Nous avons jusqu’alors présenté notre contribution pour garantir le respect d’échéances temporelles ciblées sur des tâches critiques par le biais d’un mécanisme de surveillance et de contrôle. Suite à cela nous avons proposé deux protocoles expérimentaux qui permettent de caractériser un jeu de tâches de façon à constituer un cas de test expérimental d’une part et pour configurer le-dit mécanisme d’autre part. À présent nous proposons la mise en application de tous ces éléments, par l’utilisation d’un jeu de tâches issus d’une suite de benchmark, MiBench. L’objectif est de tester et caractériser le comportement de ce mécanisme de surveillance et de contrôle sur un cas de test. Cette mise en œuvre consiste à développer un framework de configuration et d’exécution d’un jeu de tâches sur une plateforme Linux grand-public qui intègre le mécanisme ainsi que des patchs pour s’approcher d’une plate-forme temps-réel. L’objectif de ce framework est de proposer un environnement expérimental de mesures comportementales du mécanisme sur un cas de test modulable. On présentera dans un premier temps la plateforme de développement matérielle et logicielle. Nous verrons dans un second temps les détails d’implémentation du mécanisme de contrôle sur cette plateforme et comment le benchmark MiBench a été exploité dans ce contexte en application du protocole expérimental détaillé précédemment.

5.1 Plateforme de développement

5.1.1 Plateforme Matérielle

Le premier élément décisionnel repose sur le choix de la plateforme matérielle. Dans ce cadre-là, nous avons choisi une machine grand-public, un *barebone PC* basé sur un processeur Intel Core i5-8250U. Il s'agit d'un processeur à quatre cœurs, qui peuvent fonctionner à une fréquence allant de 1600 MHz à 3400 MHz. Tel qu'on peut le visualiser sur le schéma d'architecture de ce processeur en Figure 5.1, il dispose d'une mémoire locale partagée en la présence du cache L3 de 8 Mio¹. Par ailleurs, chaque cœur dispose de deux niveaux de cache individuels, L1 et L2 qui peuvent respectivement stocker 32 Kio/coeur et 256 Kio/coeur.

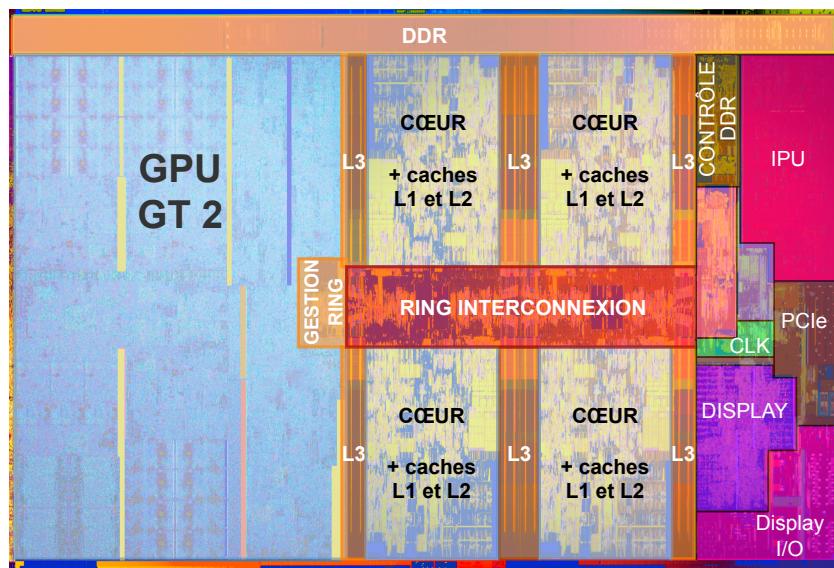


FIGURE 5.1 – Circuit Intégré annoté de l’Intel i5 8250U

Sur ce type d’architecture, de nombreux points de contention peuvent provoquer des interférences entre les tâches. Dans la version schématisée du processeur en figure Figure 5.2, une partie de ces éléments de contentions sont mis en valeur. On notera notamment le cache partagé ainsi que tous les bus de transfert de donnée (le "ring"), et les différents contrôleurs d'accès aux divers éléments. La problématique de la cohérence du cache a pu être étudié en détail notamment dans [Boniol 2019] pour les multicœurs COTS.

Dans le cadre de nos expérimentations, nous établissons la fréquence de fonctionnement du processeur à une valeur fixe de 1600 MHz. L’hyperthreading est désactivé. Il s’agit d’une technologie permettant l’exécution parallèle de code sur un même cœur par dédoublement virtuel de ce dernier. Cela fait passer le CPU de 4 cœurs physiques à 8 cœurs virtuels aux yeux du système d’exploitation. Son usage complexifie d’autant plus la gestion de l’utilisation de ressources partagées, et provoque même des partages de ressources supplémentaires, et par conséquent

1. Mébioc tet – 1 Mio = 2^{20} octets = 1 048 576 octets, bien que proche, à ne pas confondre avec 1 Mo = 10^9 octets = 1 000 000 octets.

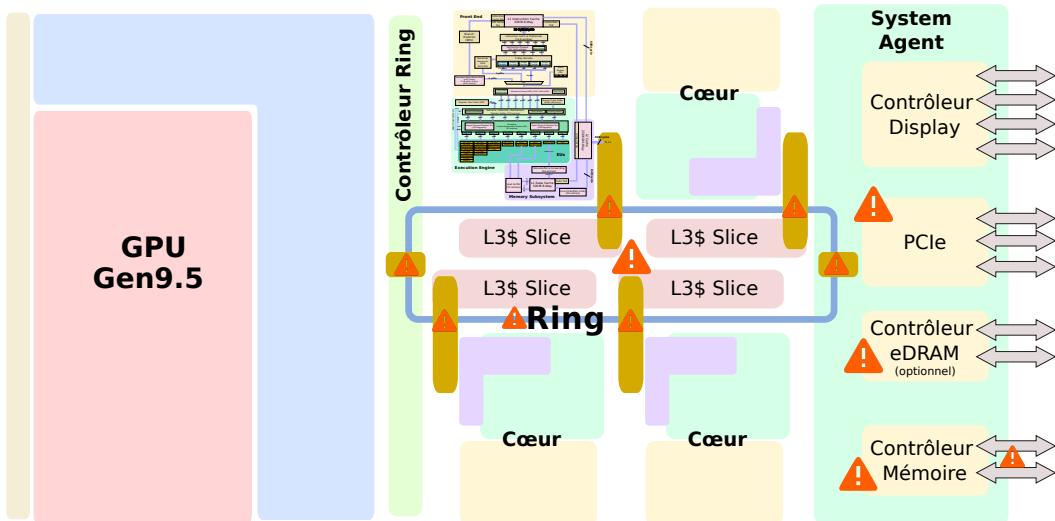


FIGURE 5.2 – Représentation schématique du processeur avec les points d’interférences notables

des sources d’interférence et donc de ralentissements d’exécution potentiels. Cela rend la maîtrise de l’exécution logicielle encore plus difficile à maîtriser par un mécanisme de contrôle et notamment pour obtenir une caractérisation représentative des interférences entre les tâches. De fait, cela nécessiterait d’observer toutes les possibilités d’exécution parallèle de code relatives à l’hyperthreading, en plus de toutes les possibilités d’exécution déjà existantes. Par conséquent, dans le cadre de notre étude et de façon plus générale, il nous semble plus pertinent de réaliser une implémentation sans utiliser de telles technologies, quitte à voir dans un second temps si leur ajout est envisageable.

Avec notre seule proposition, de nombreuses possibilités supplémentaires sont offertes en comparaison des pratiques industrielles actuelles. De ces dernières, l’on pourra citer la désactivation complète du cache qui est une pratique courante pour éviter complètement tout effet de bord non prévu par l’intégrateur du logiciel, ou encore l’usage de stratégies d’ordonnancement statiques avec des fenêtres d’exécution temporelle fixes, et potentiellement surévaluées. Ces deux éléments représentent des limitations très importantes de performance des calculateurs émergents. Ces bridages nécessaires actuellement de par un manque de maîtrise du comportement matériel, pourraient être évités avec l’utilisation de mécanismes de contrôle dynamique comme le nôtre.

Au regard de ce constat, il convient donc de réaliser ces études de façon progressive et donc de ne pas impliquer directement la totalité des mécanismes d’optimisation mis à disposition par les processeurs, qui apportent chacun une couche de complexité supplémentaire.

5.1.2 Support Logiciel

Système d’exploitation Linux – Pour mettre en application nos bancs d’essais sur cette plateforme matérielle, nous avons cherché à reprendre un système d’exploitation qui permette le plus de versatilité possible via l’utilisation d’un système Linux. L’intérêt est double.

D’une part cela offre l’opportunité de tester une grande variété de logiciels et donc de pouvoir réadapter la solution à d’autres cas et contextes d’application. Cette richesse de logiciel se traduit aussi par l’existence de bibliothèques déjà mises à disposition et éprouvées pour nos différents besoins. On pourra mentionner la capacité de gestion de l’ordonnancement et de l’exécution des tâches, l’automatisation des tests par scripts, des outils de stress du matériel ou encore de benchmarking ([King 2019]) et monitoring du logiciel comme avec [Girbal 2018]. Par ailleurs, plusieurs travaux ont déjà été effectués sur l’étude de Linux dans le cadre d’usage pour des systèmes temps-réel, tel que [Litayem 2011], [Allende 2019] ou encore [Serra 2020]. Ainsi, ce type de système d’exploitation est déjà largement utilisé pour des cas d’application comme en robotique [Bouchier 2013] ou sur des plateformes de prototypage automobile [Sivakumar 2020], [Gobillot 2018].

D’autre part, l’utilisation d’une plateforme matérielle et logicielle Linux apporte un niveau de complexité supplémentaire vis-à-vis de plateformes industrielles COTS qui emploient un micro-kernel. Cela se traduit particulièrement sur la politique d’ordonnancement et de gestion des tâches dans Linux comme cela a pu être couvert dans [Lozi 2016]. Cela présente un intérêt particulier. En effet, en obtenant des résultats probants sur l’efficacité d’un mécanisme de surveillance et de contrôle dans un environnement où l’on ne maîtrise pas totalement les couches bas niveau du logiciel alors on sera en droit d’être optimistes pour des cas d’utilisation sur des supports logiciels plus maîtrisés. La différence principale étant de soumettre un test sur une distribution Linux grand public qui implique déjà un certain nombre de tâches et de drivers qui s’exécutent en fond, qui n’ont pas été pensés pour du temps réel, plutôt que d’embarquer uniquement le logiciel dédié et contrôlé d’un cas réel.

Nous nous baserons plus spécifiquement sur une distribution Linux Mint XFCE, version 20.04, avec un noyau en version 4.15.8.

Co-noyau Xenomai – De façon à utiliser Linux dans le cadre d’applications temps-réel, nous y adjoignons Xenomai [Gerum 2004] en version 3.1. Il s’agit d’un co-noyau qui se patche au noyau de base Linux de façon à lui apporter des fonctionnalités propres au développement d’applications temps-réel. Cela apporte notamment de meilleures performances en termes de latence comparé aux appels système Linux natif. Cela est dû au fait qu’initialement, le noyau Linux présente une grande part de code qui est non-préemptif. En conséquence, l’exécution de code non critique peut retarder la gestion d’interruptions destinées à exécuter du code temps-réel. Les différentes solutions de modification du noyau comme Xenomai, ou encore le patch `preempt_rt` dans une moindre mesure, répondent directement à cette problématique pour diminuer au mieux la latence constatée. Les écarts de latence observés sur la gestion des interruptions de l’ordre de millisecondes sont réduites à des microsecondes avec Xenomai. Les différentes solutions qui modifient Linux pour tenter d’y

apporter un meilleur cadre d'utilisation de logiciel temps-réel ont pu être comparés dans [Brown 2010].

Le choix que nous avons fait a été déterminé aussi du fait qu'il fournit un framework complet pour exécuter des tâches temps-réel avec une gestion de tous les outils relatifs au domaine :

- sémaphores et mutex,
- d'envoi/réception de messages : pile, buffer, queues,
- ordonnancement et allocation de tâches avec périodicité,
- gestion d'alarmes et interruptions,
- gestion d'entrée/sortie en évitant des latences propres à Linux.

De plus, ce framework est disponible suivant différentes interfaces de développement. Cela facilite la portabilité d'une application entre différents frameworks temps-réel et le framework Xenomai natif. Ainsi, il est possible d'exécuter sur un support Xenomai du logiciel utilisant les interfaces de POSIX, PSOS, RTAI (qui est le prédecesseur de Xenomai), µ-ltron, VRTX, VxWorks et rtdm avec peu de modification du logiciel. Par ailleurs, notre choix de cette combinaison d'une carte mère basée sur un processeur Intel avec Xenomai est confortée par le fait qu'Intel ait déjà étudié la question de l'usage de Xenomai sur ses calculateurs multicœur [Intel Corporation 2009]. Ce framework sera utilisé pour l'implémentation du mécanisme de surveillance et contrôle.

De façon générale, ce qui permet à Xenomai d'offrir toutes ces fonctionnalités avec des latences réduites en cohabitation avec le noyau Linux réside dans l'ajout d'une couche **Adeos** [Gerum 2005] qui est un pipeline des interruptions. Il permet la préemption de toutes les interruptions système pour les distribuer en priorité vers le domaine Xenomai avant tout envoi de ces dernières vers le domaine Linux. Cela inclut à la fois les interruptions matérielles et logicielles, ainsi que tous les envois de signaux système (changement d'état d'une tâche pour la mettre en pause/démarrer/arrêter, gestion de mutex etc.). Cela priorise *de facto* l'exécution de code du domaine Xenomai et réduit au maximum les latences dues au noyau Linux. La différence entre un noyau Linux simple et avec un patch Xenomai est représenté en Figure 5.3. On constate que le co-noyau propose un équivalent symétrique aux composants natifs du noyau Linux, il s'agit en effet de répliquer en grande partie ce qui est mis à disposition dans le noyau Linux. À la différence d'être directement prévu pour limiter au maximum les latences d'exécution tout en ajoutant les couches nécessaires de compatibilité.

Il est à noter qu'au sein des systèmes Linux, ce que l'on nomme "*tâche*" est désigné sous le nom de processus ou de thread. La différence fondamentale entre ces deux appellations étant que chaque processus possède son propre espace mémoire virtuel, tandis que les threads peuvent se partager un espace mémoire virtuel commun s'ils sont lancés à partir d'un même processus parent. Du point de vue de l'ordonnancement du système d'exploitation, threads et processus sont équivalents. Cette différence sur la mémoire est, en ce qui nous concerne, négligeable. En effet, il s'agit bien de mémoire *virtuelle*, c'est-à-dire que du point de vue des processus ils disposent d'un espace mémoire réservé, mais dans les faits il s'agit de mémoire

physique qui est partagée. Le contrôleur mémoire du système d'exploitation étant responsable de la mise en correspondance entre adresses mémoires virtuelles et espaces mémoires physiques. Ainsi, que l'on travaille avec des threads ou des processus, les risques d'interférences par usage de ressources partagées tel que les cache reste identique. On continuera à parler de "tâches" par la suite.

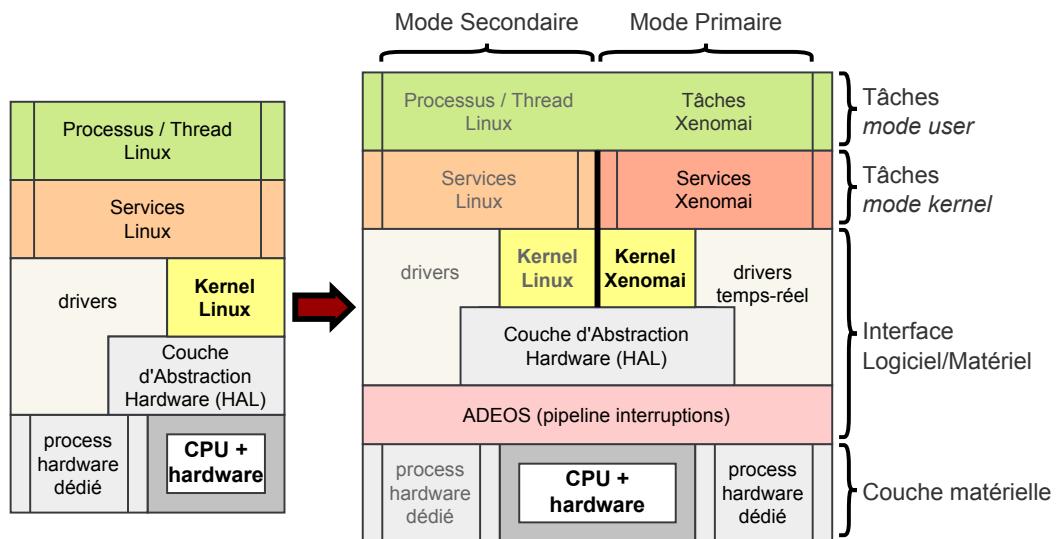


FIGURE 5.3 – Architecture du noyau Linux et Linux avec co-noyau Xenomai

Il existe deux modes d'exécution des tâches sur un système Linux : en mode User ou Noyau, que l'on peut retrouver sur les 2 couches supérieures de l'architecture logicielle de la Figure 5.3. Dans le mode Noyau, les tâches ont les droits pour exécuter des instructions privilégiées. Cette distinction est importante, car les opérations bas niveau que nous devons employer pour la surveillance et la gestion d'ordonnancement des tâches nécessite l'emploi d'instructions privilégiées. Avec l'utilisation de Xenomai s'ajoute une nouvelle dimension. Les tâches peuvent alors s'exécuter soit dans le domaine Xenomai en *mode Primaire*, soit dans le domaine Linux en *mode Secondaire*. L'avantage principal du domaine Xenomai étant la faible latence d'exécution, proche d'un système temps-réel dur. En revanche si une tâche dans ce domaine requiert des appels système Linux, alors une migration vers le domaine Linux est nécessaire. Xenomai réalise cela par un mécanisme de *shadowing* où la tâche est clonée dans le domaine Linux en mode Secondaire pour effectuer les appels systèmes nécessaires. La transition entre les modes se fait de façon "fainéante" (*lazy mode switch*) dans le sens où un changement de mode n'est fait qu'au moment où un appel de fonction requiert d'être dans l'autre mode de fonctionnement.

Si nous relevons cette spécificité logicielle, c'est parce que chaque changement de mode engendre un surcoût de temps d'exécution avec le clonage de la tâche. Par conséquent, une tâche qui utiliserait successivement des appels système Linux et des fonctionnalités Xenomai bas-niveau peut très vite engendrer un très fort surcoût d'exécution. Dans les fonctionnalités de Xenomai sont inclus la gestion de mutex, de signaux ou autres appels aux APIs temps-réel de Xenomai (gestion

d'ordonnancement, alarme, interruptions...). C'est d'ailleurs pour cette raison qu'au fil du temps un bon nombre d'appels système ont eu droit à leur équivalent dans les API de Xenomai, pour éviter un passage en mode secondaire. C'est le cas par exemple des fonctions d'écriture dans la sortie standard `printf()` qui devient `rt_printf()`. De façon plus générale, la correspondance entre les différents domaines applicatifs de Xenomai est un aspect très important de son développement pour permettre le plus facilement possible une inter-compatibilité entre les différents systèmes temps-réel et Linux. Une bonne part des enjeux et des méthodes déployées pour l'implémentation d'un système temps-réel donné sous Linux est développé dans [Gerum 2015]. Dans le cadre de l'utilisation d'un jeu de tâches arbitraire pour la réalisation de nos tests, nous devons être vigilants sur l'utilisation d'appels système et du nombre de changements de mode. Des tâches ayant de mauvais résultats sur ces métriques (notamment un surplus de changement de modes) sont probablement inadaptés à représenter des tâches temps-réel, à moins d'être modifiées spécifiquement pour éviter ce problème.

En résumé, par le biais de ce support logiciel, il est possible d'exécuter un set de tâches donné, sous forme de processus séparés. Pour chacun d'entre eux la stratégie d'ordonnancement employée, le niveau de priorité, ainsi que l'allocation de la tâche sont configurables. L'allocation indique au système d'exploitation quels sont les cœurs qui sont autorisés à exécuter une tâche, autrement dit, à quels cœurs elle est allouée. C'est ce qui permet de déterminer pour chaque tâche si elle subira un ordonnancement partitionné (allocation à un unique cœur) ou global (allocation à tous les cœurs, par défaut). Ainsi, pour un système à 4 cœurs par exemple, chaque tâche τ sera associée à une table d'affinité $A(\tau) = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$ par défaut, signifiant qu'elle peut être exécutée sur n'importe lequel des 4 cœurs. Cette liberté rend notamment service à l'ordonnanceur pour équilibrer la charge entre les cœurs. Pour notre cas, et quand il s'agit d'exécuter une application temps-réel, il est intéressant d'exploiter ce mécanisme d'affinité. De cette façon, il sera possible d'isoler des tâches critiques sur un cœur par exemple, et d'appliquer une politique semi-partitionnée (si les tâches non critiques ne sont pas allouées à des cœurs spécifiques), voire complètement partitionnée (si les tâches non critiques sont toutes aussi allouées à des cœurs spécifiques).

Et à propos d'ordonnancement sous Linux (et donc par extension, avec Xenomai), il s'établit selon un schéma assez spécifique [Ishkov 2015]. Il s'agit d'un ordonnancement global, mais qui prend en ligne de compte les 3 éléments susmentionnés dans l'ordre suivant :

- l'allocation de la tâche,
- le niveau de priorité,
- la stratégie d'ordonnancement.

En effet, il s'agit en premier lieu d'un ordonneur par niveau de priorités. Pour les tâches classiques exécutées par Linux, ce niveau de priorité est dynamique, selon le temps déjà donné à chaque tâche par le processeur. Il s'agit de l'ordonnancement *Completely Fair Scheduling* (CFS) [Wong 2008], [Pabla 2009]. En revanche, quand il s'agit de tâches à plus haut niveau de priorité (labellisées RT par le gestionnaire de tâches), où cette priorité est fixe, alors il existe plusieurs stratégies d'ordonnancement

possibles. L'ordonnanceur sélectionne les tâches en attente d'abord par niveau de priorité décroissant. Puis les tâches d'un même niveau de priorité sont alors traitées selon leur politique individuelle. On a notamment First-In First-Out (FIFO), Round-Robin (RR) et Earliest Deadline First (EDF²). Ceci étant dit, il est difficile de trancher de façon certaine sur la réaction de l'ordonnanceur en cas de coexistence de tâches de même niveau de priorité, mais avec plusieurs stratégies d'ordonnancement différentes... Ainsi, si l'on souhaite par exemple tester un système à priorité fixe on pourra laisser la stratégie d'ordonnancement par défaut et modifier uniquement les niveaux de priorité. Pour un système en round-robin, il faudra positionner toutes les tâches au même niveau de priorité, toutes avec l'ordonnancement RR.

5.2 Logiciel Applicatif

5.2.1 Benchmark MiBench

Présentation – MiBench est un benchmark qui a été développé par Guthaus & al. [Guthaus 2001] dans l'idée de proposer une librairie d'applications qui couvrent un large panel de domaines. Nous l'utilisons de façon à pouvoir en sélectionner des tâches pour représenter au mieux un cas d'application réaliste. Ce benchmark dispose de codes source ainsi que de données type qui sont utilisables pour lancer chaque application. Aussi, chacune d'entre elle est disponible en deux versions, "petite" et "grande" qui, comme cela le laisse penser, implique des temps d'exécution ou une empreinte mémoire (selon l'application) plus ou moins conséquente. Cela se fait soit par l'utilisation d'un code différent volontairement plus complexe et plus demandeur en ressources de calcul, soit par la modification des données d'entrée/sortie pour prévoir des plus grandes tailles de données (quand il s'agit d'un traitement d'image par exemple). Au total, ce sont 30 tâches, chacune en version "petite" et "grande" donc. Le résumé des tâches de ce benchmark est présenté dans la Tableau 5.1 .

TABLE 5.1 – Tâches MiBench

Nom	Description	Type d'entrée	Type de sortie
basicmath	calculs scientifiques	/	données (dec.)
bitcount	comptage de bits vers entiers	texte ASCII	texte ASCII
qsort	algorithme de tri	texte ASCII	texte ASCII
susan c	reconnaissance de coins	image (.pgm)	image (.pgm)
susan e	reconnaissance de bords	image (.pgm)	image (.pgm)
susan s	lissage d'image (réduction de bruit)	image (.pgm)	image (.pgm)
jpeg c	encodeur JPEG	image (.ppm)	image (.jpeg)
jpeg d	décodeur JPEG	image (.jpeg)	image (.ppm)
lame	encodeur MP3	audio (.wave)	audio (.mp3)
mad	décodeur audio MP3	audio (.mp3)	audio (.wave)
tiff2bw	conversion en noir et blanc	image (.tiff)	image (.tiff)
tiff2rgba	conversion couleurs RGB en TIFF	image (.tiff)	image (.tiff)

2. g-EDF pour être exact, disponible par patch [Lelli 2011]. Linux ne gère pas une allocation partitionnée d'une tâche ordonnancée par EDF.

TABLE 5.1 – Tâches MiBench (suite)

Nom	Description	Type d'entrée	Type de sortie
tiffdither	tramage noir et blanc (dithering)	image (.tiff)	image (.tiff)
tiffmedian	réduction de plage de couleur	image (.tiff)	image (.tiff)
dijkstra	recherche de plus court chemin	texte ASCII	texte ASCII
patricia	recherche sur arbre PATRICIA	texte ASCII	texte ASCII
ghostscript	interpréteur PostScript	PostScript	image (.ppm)
ispell	Vérificateur orthographique	texte	texte ASCII
rsynth	synthèse vocale de texte	texte	audio (.AU)
stringsearch	recherche de mot dans un texte	texte	texte
blowfish d	déchiffrement blowfish	données (bin.)	données (bin.)
blowfish e	chiffrement blowfish	texte	données (bin.)
pgp d	déchiffrement asymétrique	données (bin.)	texte
pgp e	chiffrement asymétrique	texte	données (bin.)
rijndael d	déchiffrement AES	données (bin.)	texte
rijndael e	chiffrement AES	texte	données (bin.)
sha	algorithme de calcul de hash	texte	texte ASCII
adpcm c	encodeur de PWM	audio (.wave)	données (bin.)
adpcm d	décodeur de PWM	données (bin.)	données (bin.)
CRC32	somme de contrôle 32 bits	audio (.wave)	texte ASCII
fft (fft ⁻¹)	transformée de fourrier (/inverse)	signal (sinus)	signal (sinus)
gsm toast	encodage GSM	audio (.AU)	données (bin.)
gsm untoast	decodage GSM	données (bin.)	audio (.AU)

Ces tâches sont classifiées en 6 domaines d'application : automobile, réseau, utilisateur, bureautique, sécurité et télécoms. On retrouve ici la notion de criticité mixte qui nous intéresse pour la cohabitation de ces fonctionnalités au sein d'un même calculateur, car chaque domaine applicatif implique sans aucun doute des niveaux de criticité différents. Il faut cependant préciser que nous n'avons pas pu faire fonctionner toutes ces fonctions, à cause de difficultés de compilation difficilement surmontables de par la complexité du code impliqué notamment. Par conséquent, un premier tri de ces tâches s'effectue naturellement.

Adaptation — Aussi, il a fallu modifier le benchmark manuellement pour convenir à une utilisation en tant que librairie de notre framework. Les codes sources ont été modifiées un-par-un de deux façons :

- Uniformiser la méthode de récupération des données d'entrée de chaque application, de même que pour la sortie,
- Proposer une fonction appelable depuis du code externe pour lancer un job de chaque tâche MiBench.

Avec ces modifications, il a été possible de compiler chaque tâche en tant que fonction en librairie, qui sera reliée par la suite à notre framework expérimental pour être utilisé.

Le récapitulatif des tâches que nous avons pu exploiter à l'issue de tout cela, classées par domaines, est listé en Tableau 5.2. Des analyses comportementales de ces tâches ont déjà pu être réalisées par le passé, notamment par Blin & Al. [Blin 2016b] sur la consommation mémoire et le profil d'accès mémoire/calculs/écriture mémoire de ces tâches. Ce type de données et les expérimentations réalisées pour les obtenir demeurent donc une source précieuse et complémentaire au protocole expérimental que nous proposons pour la caractérisation des tâches et la compréhension de leur rôle dans l'apparition d'interférences.

TABLE 5.2 – Tâches MiBench conservées

Domaine	Tâches
Automotive	basicmath, bitcount, qsort, susan (smooth/edges/corners)
Network	dijkstra, patricia
Consumer	jpeg (décode/encode)
Office	stringsearch
Security	blowfish (décode/encode), rijndael (décode/encode), sha, CRC32
Telecom	adpcm (décode/encode), FFT, FFT ⁻¹ , gsm (décode/encode)

5.2.2 Charge de test

En complément du benchmark pour représenter la charge utile du système, il nous faut l'outillage nécessaire à l'application d'un stress du processeur pour obtenir les profils des tâches en pire cas tel que décrit dans le Chapitre précédent. À cette fin, nous utilisons deux éléments.

D'une part la suite **Stress-ng** [King 2019] qui est un outil relativement complet et avancé pour stresser chaque partie du système de façon assez précise. Il est ainsi possible de lancer une charge artificielle forcée sur l'utilisation du CPU, sur le cache (avec même des effacements forcés du cache), la mémoire ou encore les entrées/sorties (communication réseau par exemple). Pour pousser le système dans des conditions de fonctionnement encombrées, nous utiliseront en particulier la commande suivante :

```
stress-ng --ionice-class rt --cache 4 \
--fault 4 --io 4 --matrix 2
```

Si l'on regarde en détail cette commande, on identifie en premier lieu le passage en priorité haute avec **--ionice-class rt** de façon à ce que le stress ne reste pas relégué à l'arrière plan best-effort. Ensuite, 4 éléments de stress distinct sont lancés :

- cache 4** – Stress du cache via 4 processus,
- fault 4** – Stress de la mémoire (par provocation de *page faults*),
- io 4** – Stress des entrées/sorties génériques,
- matrix 2** – Stress multi-éléments sur l'utilisation CPU, cache et mémoire par calculs matriciels avec virgule flottante via 2 processus.

En complément de cette outil, Xenomai propose aussi de quoi effectuer des stress avec la fonction `DoHell`. En revanche, il est moins clair des effets exacts de ce stress sur le système. Par conséquent, nous pourront employer les deux pour avoir une double soumission au stress de nos tâches. Nous pourront donc de la même façon utiliser :

```
dohell -b -s 192.168.0.1 -m /tmp 800
```

Cette commande permet d'exécuter une charge spécifique avec l'option `-b`, en plus d'un fort trafic réseau avec `-s 192.168.0.1` et une charge de lecture/écriture mémoire avec `-m /tmp`, pendant 800 secondes.

5.2.3 Implémentation de la plateforme expérimentale logicielle

À partir de tous les éléments que nous venons de présenter, il est possible de mettre en place une plateforme expérimentale complète pour tester le mécanisme de surveillance et contrôle dans un environnement personnalisable. Une telle plateforme se caractérise par 3 grands leviers :

- le **support de développement**, tant matériel que logiciel que nous venons de présenter ;
- les **hypothèses de modélisation**, que nous avons présenté au chapitre 3. Il s'agit du modèle de tâches à criticité duale et de chaîne de tâches ainsi que le modèle du mécanisme de surveillance et de contrôle étudié ;
- les **paramètres d'entrée**, qui se compose d'une part du système logiciel auquel on applique les hypothèses de modélisation (en l'occurrence, les tâches MiBench) et d'autre part du paramétrage du mécanisme. Il s'agit donc des éléments présentés dans le chapitre 4.

L'ensemble de ces éléments sont agrégés dans la Figure 5.4. Cela permet de visualiser simplement les différents éléments d'entrée de la plateforme expérimentale, qui sont tout autant de vecteurs ajustables pour tester l'incidence des différents paramètres de la plateforme.

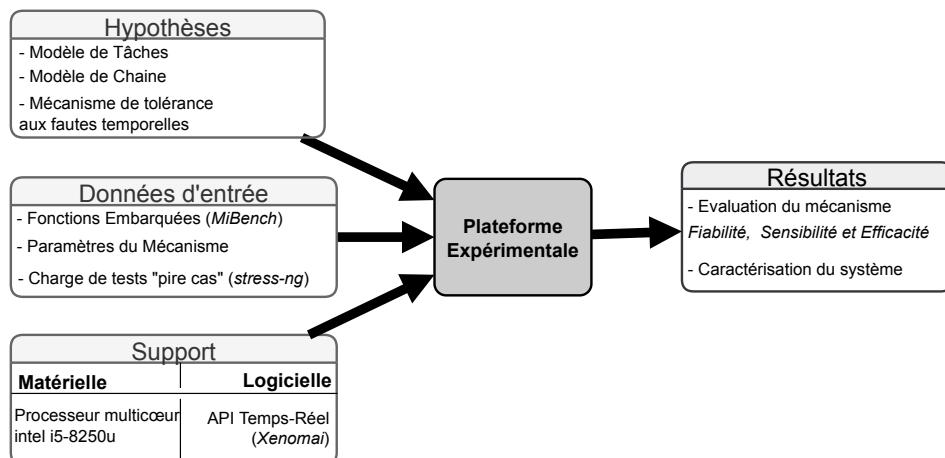


FIGURE 5.4 – Structure générale de la plateforme expérimentale

5.2.3.1 Framework expérimental

La mise en place de cette plateforme expérimentale a débutée par la configuration, compilation puis installation d'un noyau Linux 4.15.8 patché avec un co-noyau Xenomai. Ensuite il a fallu mettre en place un framework générique de lancement de tests. Le framework en question répond aux objectifs suivants :

- récupérer une liste de tâches à exécuter avec leurs paramètres associés suivant un fichier de configuration,
- lancer les tâches paramétrées, avec une encapsulation par le mécanisme de Surveillance,
- lancer une tâche dédiée au mécanisme de Contrôle,
- exécuter les tâches sur une durée déterminée puis sauvegarder toutes les traces d'exécutions dans un fichier de log.

Pour réaliser cela, on implémente un module `TaskLauncher` de configuration, qui prend en donnée d'entrée un fichier `conf.in` qui indique des groupes de tâches à lancer.

Ce groupement (plutôt que de lister directement toutes les tâches) permet de distinguer les tâches non critiques des tâches de la chaîne critique. Chaque ligne du fichier indique donc un groupe de tâches, avec un identifiant ID associé. ID = 0 pour des tâches non critiques, et ID ≥ 1 indique une chaîne critique et donc l'identifiant de la chaîne. En plus du niveau de criticité, il inclut la stratégie d'ordonnancement du groupe de tâches ainsi qu'un identifiant de groupe. S'il s'agit d'un groupe désignant une chaîne de tâches, l'échéance bout-en-bout est aussi indiquée. Un exemple de fichier de configuration est indiqué dans l'encadré de Code 5.1.

1	NAME	ID	PATH	DEADLINE (ms)	SCHED_POLICY
2	NonCritical	0	./bestEffort.in	99999	BE
3	ComputeTraj	7	./ComputeTraj.in	187	RM
4	<code>// ID = 0 : deadline value is unused.</code>				

Code 5.1 – Exemple type de fichier d'entrée `conf.in`

Aussi, chaque groupe dans ce fichier de configuration pointe vers des fichiers de configuration secondaires `tasks.in` qui donnent la liste de toutes les tâches de ce groupe à être exécutées avec les paramètres de chacune d'elles. Un exemple d'un tel fichier de configuration secondaire est donné en Code 5.2.

1	ID	name	rWCRT	T	P	A	prec	FUNC	ARGS
2	1	fft_S	129000	40	10	1	0	fft	8 2048 [...]
3	2	bitcount_S	93000	60	10	1	1	bitcnts	75000 [...]
4	3	basicmath_S	68000	40	10	1	2	bmath_S	> ./out/[...]
5	4	sha_S	49000	60	10	1	3	sha	< ./dat/[...]
6	5	fft_inv_S	25000	40	10	1	4	fft_i	8 4096 >[...]
7	6	gsmUToast_S	94000	100	10	1	5	utoast	-dfps -c [...]
8	7	fft_S	67500	100	10	1	6	fft	8 16384 >[...]
9	8	patricia_L	0	100	10	1	7	patricia	< ./dat/[...]
10	<code>// T=Period P=Priority A=Affinity prec=Precedency</code>								
11	<code>// rWCRT in us ; T in ms ; A in hexadecimal</code>								

Code 5.2 – Fichier d'entrée pour une chaîne de tâche

Dans le détail, ces fichiers de paramètres des tâches comportent les éléments suivants, qui sont traités par le module **TaskManager** : ([param.] = optionnel)

- id** et **name** – identifiant unique et nom de la tâche,
- function** – pointeur vers la fonction à réaliser (sous forme de librairie),
- [**arguments**] – paramètres propres à la tâche et pointeur vers données d'entrée,
- affinity** – affinité, c.-à-d. les coeurs sur lesquels la tâche peut être exécutée,
- periodicity** – périodicité d'exécution des jobs (en millisecondes)
- priority** – niveau de priorité (si priorité fixe) (entre 0 et 100 sous Linux),
- [**precedency**] – (si critique) : tâche précédente dans la chaîne de tâches,
- [**rWCRT**] – (si critique) : paramètre de l'Agent de Contrôle $rWCRT(\tau_i)$ en μs .

On peut remarquer 2 informations intéressantes dans le fichier de configuration d'une chaîne de tâche. D'abord à la ligne 2, la valeur de **prec = 0** pour désigner la tâche précédente. Cela indique de façon explicite qu'il s'agit de la tâche d'entrée de la chaîne. De la même façon, à la ligne 9, la valeur de pire temps de réponse restant **rWCRT = 0** indique qu'il s'agit de la tâche de sortie de la chaîne.

Les processus principaux de notre framework expérimental sont représentés avec leurs interactions en Figure 5.5. Une fois les fichiers de configuration traités et interprétés par le **TaskManager**, ce dernier peut faire appel aux APIs Xenomai pour lancer toutes les tâches du système. Cela va donc inclure toutes les tâches décrites dans les fichiers de configuration, mais aussi une tâche dédiée à l'Agent de Surveillance et Contrôle. Une fois l'ensemble des tâches créées avec l'encapsulation des tâches et les canaux de communication dédiées au mécanisme de Surveillance, la totalité du système est lancé pour la durée fixée de l'expérimentation.

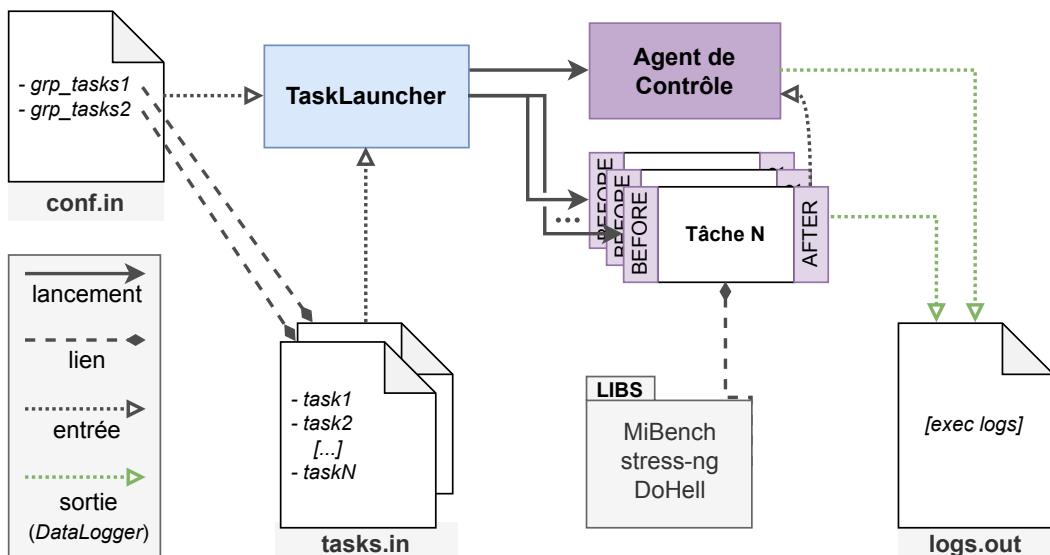


FIGURE 5.5 – Fonctionnement du framework expérimental

À la fin de l'expérimentation, le Mécanisme de Contrôle gèle l'exécution de l'ensemble de tâches et enregistre les profils d'exécution de la chaîne de tâche dans un fichier de sortie `logs.out`. Enfin, grâce à un `DataLogger` ajouté dans l'encapsulation des tâches, chacune d'entre-t-elle enregistre aussi en sortie tous ses logs d'exécution. Ce module de log d'exécution, propre à notre plateforme expérimentale, est décorrélé du mécanisme de Surveillance et Contrôle. Il a été ajouté uniquement pour obtenir des mesures de début et fin d'exécution plus complètes (incluant les tâches non critiques), de façon à analyser plus en détail le comportement du système.

5.2.3.2 Agent de Surveillance et Contrôle

L'agent de Surveillance et Contrôle que nous avons développé se décompose en 2 processus. D'une part un processus de réception des données de fin d'exécution des tâches critiques, et d'autre part un module de prise de décision pour effectuer le Contrôle à proprement parler.

L'interaction entre les modules du Core Control Component et du Task Wrapper Component est illustré sur la Figure 5.6, avec la réception asynchrone des timestamps de fin de tâches critiques (Real-Time) et l'envoi de signal de passage en mode dégradé vers les tâches non critiques (BE - Best-Effort) en cas d'anticipation.

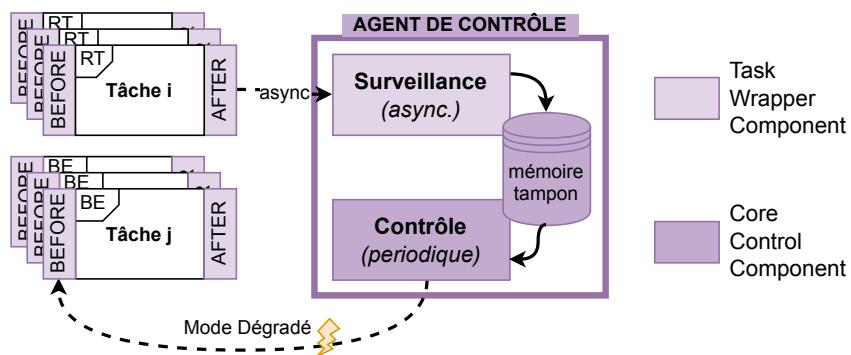


FIGURE 5.6 – Interactions Agent de Surveillance-Contrôle avec les tâches

Ainsi, le module de Surveillance reçoit les données de début/fin des tâches critiques de façon asynchrone, pour enregistrer ces données dans une mémoire tampon. Ensuite, à chaque période T_{CCC} , le module de Contrôle traite les données en mémoire tampon pour mettre à jour l'État de la chaîne de tâches et calcule la condition Équation 3.1 - $RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW} \leq D_c$ pour anticiper une situation à risque qui demande un passage en mode dégradé. Nous avons fait ce choix d'implémentation en deux composantes de façon à ce que la liaison entre l'Agent de Contrôle et les Wrappers se résume à un envoi de message asynchrone, avec une tâche dédiée très simple qui a pour seul rôle de mettre ces messages en mémoire tampon pour être traités périodiquement par le CCC. Ainsi, une erreur dans l'envoi de ces messages ou dans leur réception n'empêche pas au module de Contrôle d'effectuer la vérification périodique. Cela évite aussi le partage de ressources entre les Wrappers et l'Agent de Contrôle. Et l'unique ressource partagée est interne à ce dernier, en la présence de la mémoire tampon. En conclusion, la modélisation de l'ensemble figure dans le diagramme de classe simplifié en Figure 5.7.

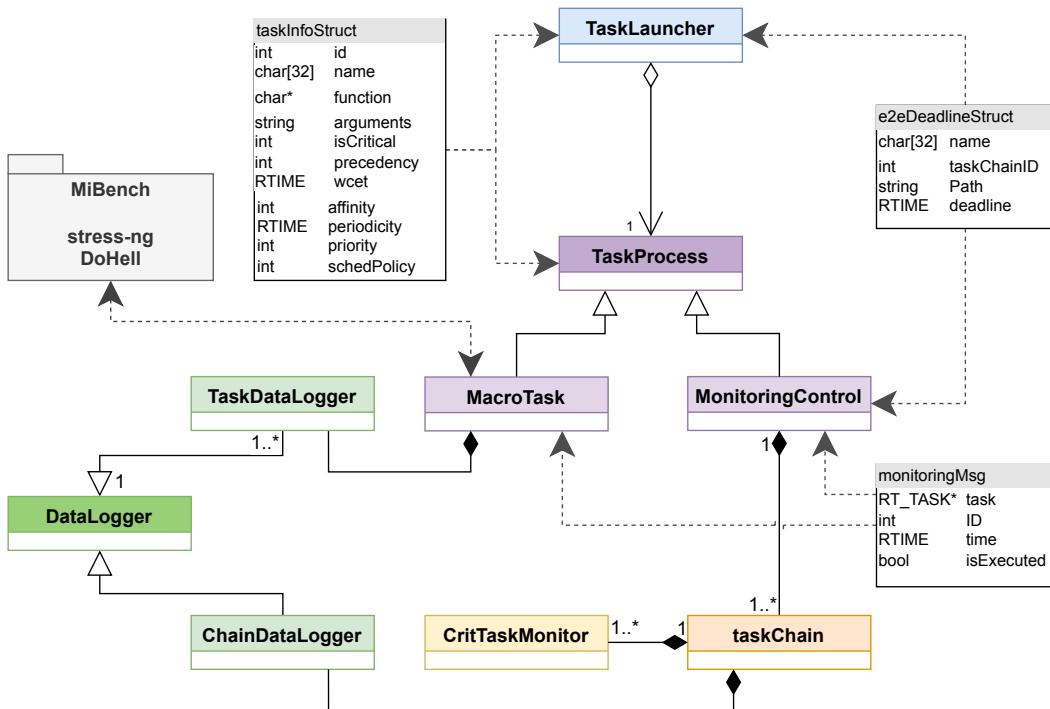


FIGURE 5.7 – Architecture Logicielle du Framework

Sur ce diagramme de classe simplifié figure en bleu le module de mise en place et de lancement expérimental décrit précédemment, en violet les classes qui permettent de lancer des tâches (que ce soit l'agent de contrôle ou les tâches à exécuter, critiques et non critiques), avec en orange les structures mémoire de gestion de l'État de la chaîne de tâche et des tâches critiques, et pour finir en vert les modules supplémentaires d'enregistrement des logs détaillés d'exécution pour être enregistrés, que ce soit pour la chaîne de tâche et pour les tâches individuelles. On retrouve aussi les structures de gestion des fichiers de configuration d'entrée, ainsi que la structure des messages asynchrones échangés entre les tâches critiques et l'Agent de Surveillance.

Tous les éléments de la plateforme expérimentale ont ainsi été présentés, avec les choix que nous avons adoptés. Nous présentons dans la suite son exploitation par la mise en application du protocole expérimental global présenté au chapitre 4.

5.3 Application du Protocole Expérimental à MiBench

L'utilisation de MiBench de façon à être au plus proche d'un cas d'utilisation réel qui embarquerait des logiciels à criticité mixte présente à la fois des avantages et inconvénients. Il permet d'un côté d'exploiter directement un jeu de tâche déjà reconnu et prêt à l'emploi, sur lequel les comportements ont été étudiés (utilisation de cache, charge CPU, entrées/sorties...). En revanche, il n'a pas été programmé initialement dans le but d'être exploité dans le cadre d'un système ne serait-ce que partiellement temps-réel. Par conséquent, la façon dont les tâches sont codées peut ne pas correspondre parfaitement à nos besoins, notamment via des incompatibilités

entre les stratégies d'ordonnancement temps-réel et les appels de fonctions internes à ces programmes. Par conséquent, il est important d'appliquer le protocole d'implémentation de façon à caractériser correctement ce jeu de tâches au sein de notre framework. Cela permet d'en exclure les éventuelles tâches qui ne sont pas adaptées. Comme nous l'avons mentionné au chapitre précédent, il y a particulièrement 3 raisons fondamentales qui vont justifier l'exclusion d'une tâche :

- de trop grands écarts en temps d'exécution comparé au reste des tâches
(*par exemple une tâche qui s'exécute pendant plusieurs centaines de millisecondes, contre la dizaine de millisecondes pour les autres*)
- l'utilisation répétée de fonctions qui forcent des changements de domaine d'exécution entre Linux et Xenomai, cela provoque des délais supplémentaires variables et casse potentiellement l'ordonnancement,
- une trop forte irrégularité de comportement,
(*par exemple une tâche qui successivement dans les mêmes conditions apparentes peut avoir un temps d'exécution allant du simple au décuple*),

De plus, même pour les tâches conservées ces données sont utiles, car elles donnent des pistes pour identifier si ces tâches ont plutôt un profil de tâche non-critique type "Best-Effort", ou bien de tâche critique temps-réel.

5.3.1 Phase de Conception

Dans un premier temps, nous appliquons les étapes de caractérisation et conception présentées dans le protocole général illustré en Tableau 4.4. Elles correspondent à la caractérisation des tâches de façon individuelle, pour constituer le cas de test (chaîne de tâches critiques et tâches non critiques) et la caractérisation de la chaîne de tâches critiques.

5.3.1.1 Profil des tâches individuelles (① & ②)

Tout d'abord pour la caractérisation des tâches MiBench listées en Tableau 5.2, elles sont exécutées individuellement (en isolation) pour une durée de test de 120 secondes, d'abord à une période de 100ms entre chaque job. Le test est effectué à nouveau mais accompagné de la charge de stress du calculateur tel que présenté précédemment. À l'issue de ces 2 mesures exécutées sur chacune des 23 tâches MiBench, chacune en version "Petite" et "Grande", cela donne $2 * 23 * 2 = 92$ expérimentations qui donnent les résultats détaillés de temps d'exécution ainsi que le nombre de changements de mode domaine et d'appels systèmes. Nous avons testé pour la charge de stress à la fois `stress-ng` et `doHell` et il s'avère que les résultats sont relativement similaires.

À l'issue de cela, on a pu classifier les tâches en 4 catégories, suivant leur sensibilité relative au stress (écart entre les temps d'exécution en isolation et sous stress), leur durée moyenne d'exécution, et le nombre de changements de modes :

- CLEAN – les tâches sont peu influencées par le stress, peu voire pas d'appels systèmes et changements de modes, temps d'exécution dans la moyenne.

- NOISE – sensibilité au stress notable, temps d'exécution plutôt au-dessus de la moyenne avec du stress.
- OVERHEAD – très forte sensibilité au stress, avec des appels systèmes en plus qui donnent des temps d'exécution très élevés en stress.
- REJECT – Comportement erratique, avec très fortes variations de temps d'exécution, ou bien temps d'exécution à un ordre de grandeur différent comparés à la moyenne.

Ce tri des tâches est résumé dans la Tableau 5.3. Aux vues des résultats, nous excluons les tâches REJECT qui ont un comportement trop éloigné de nos critères. Les tâches OVERHEAD pourraient servir de tâches non critiques, mais ce sera à employer avec précaution et de façon très limitée à cause des temps d'exécution élevés qui risquent de rendre le système non ordonnable très facilement. Les tâches NOISE pourront servir de tâches non critiques et au sein de la chaîne de tâches critiques dans une certaine mesure pour y apporter la sensibilité au stress que l'on souhaite prévenir. Enfin, les tâches CLEAN semblent parfaitement correspondre en tant que tâches temps-réels qui représentent des blocs de code peu sensibles au stress.

TABLE 5.3 – Tâches MiBench classifiées

Domaine	Tâches
CLEAN	fft-P, fft-G, fft-inv-P, basicmath-P, basicmath-G, bitcount-P
NOISE	jpeg-D-P, susan-corners-P, susan-smooth-P, susan-edges-P, sha-P, gsm-Toast-P, gsm-UToast-P, fft-inv-G, rijndael-E-P, adpcm-C-P, sha-G, susan-corners-G, susan-edges-G
OVERHEAD	stringsearch-G, jpeg-C-P, rijndael-D-P, jpeg-D-G, jpeg-C-G, adpcm-D-P, susan-smooth-G
REJECT	rijndael-D-G, adpcm-D-G, gsm-Toast-G, adpcm-C-G, gsm-UToast-G, rijndael-E-G (<i>trop longues</i>), qsort-P, qsort-G, patricia-P, patricia-G, blowfish-E-P, blowfish-D-G, blowfish-D-P, blowfish-E-G (<i>trop courtes</i>)

(-P/-G = version Petite/Grande ; -D/-E = Decode/enCode)

Le détail des résultats de temps d'exécution obtenus lors de la caractérisation de chacune des tâches MiBench figure dans le Tableau A1 en Annexe. Avec ces données, on a constaté que la majorité des temps d'exécution des tâches MiBench étaient de l'ordre de la dizaine de millisecondes, allant de 2-3ms pour les plus courtes à 30-40ms pour les plus longues. On a aussi pu constater un nombre de changements de domaine d'exécution "de base" de 58 changements, qui sont dû à la phase de configuration de l'expérimentation et à la phase de terminaison avec l'écriture des logs. Ainsi, des tâches comme adpcm-C-G (version Grande, option enCodage) avec un temps d'exécution moyen de 432 ms sont rejetées. Notons qu'à l'inverse certaines tâches comme qsort ont été écartées parce qu'elles avaient un temps d'exécution bien trop court, de l'ordre de microsecondes. Cela n'étant pas normal, car ces tâches présentaient des temps d'exécution plus élevés quand utilisées en dehors de notre

framework (version non modifiée). On suppose donc une incompatibilité avec notre framework de test qui n'a pas pu être levée.

5.3.1.2 Profil de la Chaîne de tâches (③ & ④)

À partir de la caractérisation des tâches, on spécifie la chaîne de tâches critiques, composée de 5 tâches :

$$FFT \rightarrow Bitcount \rightarrow Basicmath \rightarrow FFT^{-1} \rightarrow sha.$$

Au regard des profils de temps d'exécution des tâches mesurés précédemment, le fichier d'entrée décrivant l'ensemble des paramètres de la chaîne est déclaré comme suit :

ID	pre	name	FUNC	P	T	rWCRT	A	ARGS
1	0	fft_S	fft	10	40	xx	2 8	2048 >[output]
2	1	bitcount_S	bitcnts	10	60	xx	2	75000 >[output]
4	2	bmath_S	math_s	10	40	xx	2	>[output]
6	4	fft_inv_S	fft	10	40	xx	2 8	4096 -i >[output]
7	6	sha_S	sha	10	60	xx	2	<[input] >[output]

Code 5.3 – Chaine de tâche sélectionnée

Bien entendu, avant la phase ③ de caractérisation de la chaîne de tâches critiques en l'exécutant en isolation, la colonne des $rWCRT(\tau_i)$ est non renseignée.

La chaîne de tâches critiques est exécutée dans deux conditions différentes, en isolation à l'image du mode dégradé³ (c.f. étape ③), et avec un stress imposé via stress-ng (c.f. étape ④). Cela permet d'une part de déterminer une échéance d'exécution bout-en-bout en adéquation avec le profil d'exécution de la chaîne, et de vérifier la pertinence de tester notre mécanisme en constatant une influence des interférences sur les temps de réponse bout-en-bout.

TABLE 5.4 – Profil de la chaîne de tâches critique - isolée et avec stress-ng

	temps en isolation			temps sous stress		
	Min	Moyen	Max	Min	Moyen	Max
temps de réponse (ms)	90.11	125.85	130.06	90.18	164.00	306.87

Sur 120 secondes d'expérimentation, on obtient 1985 exécutions bout-en-bout de la chaîne de tâche en isolation, de même avec le stress imposé. On obtient alors un profil d'exécution, qui agrège tous les temps de réponses qui ont été mesurés. Le profil obtenu est représenté dans la Figure 5.8. Au regard de ces mesures, on identifie le temps d'exécution nominal de la chaîne de tâche qui est autour des 130 ms et on peut alors définir une échéance de temps de réponse bout-en-bout arbitraire $D_C = 200\text{ ms}$, près du double du temps d'exécution bout-en-bout. Les détails des mesures de temps de réponse moyen et maximum en isolation et avec le stress forcé sont indiqués dans la Tableau 5.4. On confirme la sensibilité de la

3. **Mode Dégradé** – Pour rappel, le mode dégradé correspond, avec notre mécanisme, à l'arrêt temporaire des tâches non critiques pour exécuter les tâches critiques en isolation sur un cœur dédié.

chaîne à des interférences forcées, avec un pire cas à 300 ms de temps de réponse bout-en-bout. De façon générale, on remarque un glissement du temps d'exécution moyen de 40 ms supplémentaires, ce qui correspond à une période des tâches à la fréquence d'occurrence la plus forte.

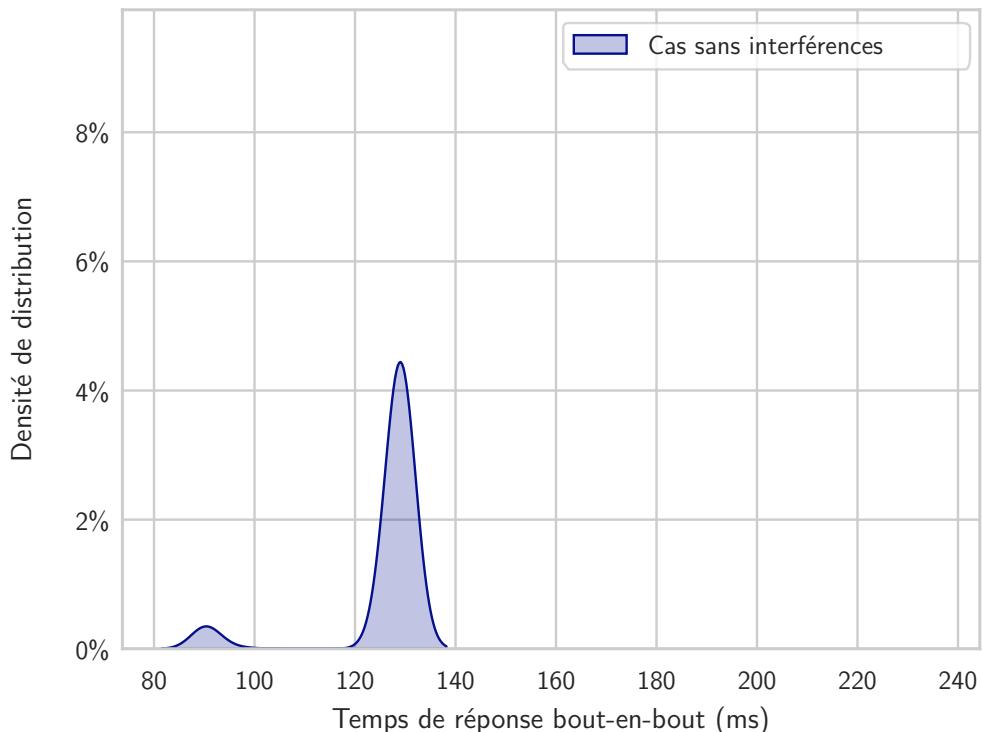


FIGURE 5.8 – Profil de la chaîne de tâche en fonctionnement nominal sans mécanisme de contrôle

5.3.2 Phase de Configuration du mécanisme

Cette phase est dédiée à la configuration des paramètres du Core Control Component, c'est-à-dire les valeurs de t_{sw} le temps de passage en mode dégradé et W_{max} la durée maximale entre 2 points de contrôle, mais aussi les $rWCRT(\tau_i)$ qui sont les pires temps d'exécution restants garantis en mode dégradé, selon les tâches de la chaîne qu'il reste à exécuter.

5.3.2.1 Paramétrage de l'agent de contrôle (③ et ④)

Constantes d'exécution du CCC En ce qui concerne les constantes de fonctionnement du Core Control Component, qui sont pour rappel t_{SW} et W_{max} , les choses sont relativement simples. Tout d'abord, pour la période de fonctionnement de l'Agent de Contrôle, au regard de la chaîne de tâches choisie la période maximale au plus lent qu'il serait possible d'employer serait toutes les 40 ms, avec une mémoire

tampon de 5 timestamps. Il s'agit respectivement de la période la plus courte des tâches de la chaîne et du nombre de tâches dans la chaîne, pour garantir qu'aucun timestamp d'exécution des tâches critiques ne soient perdus.

Cependant, de façon à mesurer l'impact de l'exécution du CCC sur l'utilisation des ressources de façon plus conséquente, et pour avoir des points de contrôle plus proches les uns des autres (et donc une meilleure réactivité du mécanisme), nous fixons $W_{max} = 2$ ms. Autrement dit toutes les 2 ms, le module de contrôle mettra à jour l'État courant de la chaîne de tâches critiques et recalculera l'inégalité d'anticipation du risque de dépassement d'échéance. Avec les tests en situation nominale du jeu de tâches complet, on pourra mesurer le temps d'exécution total du CCC et donc son empreinte sur le système, qui devra être la plus limitée possible.

Pour finir, il reste à déterminer le temps de passage en mode dégradé t_{SW} . Pour cela, on va mesurer le temps qu'il faut entre l'instant où le CCC prend la décision de passer en mode dégradé et le moment où toutes les tâches non-critiques ont reçu le signal de stop. De même que pour la mesure de l'empreinte du mécanisme, cela se fera dans l'étape ⑤ où le système est lancé avec la totalité des tâches (critiques et non critiques) sélectionnées, avec la surveillance (réception des timestamps) mais sans le Contrôle.

Pire Temps d'Exécution restants en mode dégradé En complément du test précédent effectué sur la chaîne de tâches critiques en isolation, les paramètres de l'Agent de Surveillance et Contrôle peuvent être définis. En effet, lors de la phase ③ en isolation, on enregistre de façon détaillée les dates de début et fin des tâches critiques. Cela sert au mécanisme de reconstituer les exécutions de chaînes avec la contrainte de précédence. Cette trace nous est aussi utile pour estimer les valeurs de $rWCRT(\tau_i)$ nécessaires au fonctionnement du mécanisme. En effet, avec cette trace d'exécution, on peut rec算uler les durées d'exécution restantes à partir des dates de terminaison de chacune des tâches jusqu'à la fin de la chaîne, tel qu'illustré dans le schéma 5.9.

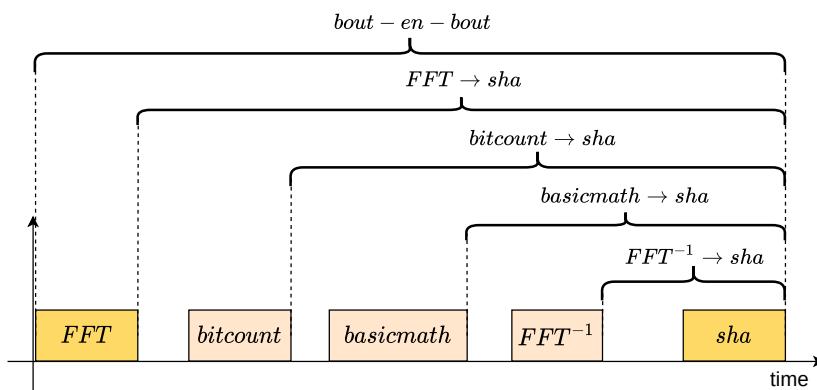


FIGURE 5.9 – Mesures des temps d'exécution restant sur une Trace d'exécution

On obtient ainsi 5 mesures de temps d'exécution restant constaté pour chaque instance p de la chaîne qui s'est exécuté tout du long de l'expérimentation. L'agrégation de ces données est résumée dans le Tableau 5.5. Il est alors possible de

déterminer les valeurs de $rWCRT(\tau_i)$, comme étant une borne maximum du temps d'exécution des tâches restantes de la chaîne de tâche, tel qu'enregistrée par le module de surveillance. Ainsi, les valeurs des paramètres d'anticipation $rWCRT()$ choisis sont indiqués dans la dernière colonne du Tableau 5.5.

TABLE 5.5 – Profil d'exécution de la chaîne de tâches critiques en mode dégradé

	Exécution restante	Temps moyen	Temps max	$rWCRT()$
bout-en-bout		125.86 ms	138.26 ms	/
FFT → sha		103.79 ms	128.25 ms	129 ms
bitcount → sha		77.58 ms	96.61 ms	97 ms
basicmath → sha		59.82 ms	85.24 ms	86 ms
FFT ⁻¹ → sha		27.06 ms	42.91 ms	43 ms

5.3.2.2 Vérification du jeu de tâches complet (⑤)

En complément de la chaîne de tâches critiques il a bien sûr fallu faire une sélection de tâches pour représenter la partie non critique du système et en fixer l'allocation sur les coeurs, les périodes d'exécution, etc. Le choix s'est naturellement porté sur des tâches classifiées NOISE, mais aussi quelques-unes classées CLEAN tel qu'on peut le voir sur le fichier de configuration 5.4 :

ID	name	FUNC	P	T	A	ARGS
11	BE_sha_S	sha	0	80	2	< [input] > [output].txt
12	BE_susan-S_corn	susan	0	80	2	-c > [input].pgm < [output].pgm
13	BE_susan-S_smoo	susan	0	80	2	-s > [input].pgm < [output].pgm
14	BE_susan-S_edg	susan	0	80	2	-e > [input].pgm < [output].pgm
15	BE_cjpeg-S	cjpeg	0	30	F	-dct int -progressive -opt [...]
16	BE_gsmToast_S	toast	0	60	4	-fps -c [input].au > [output]
17	BE_gsmUToast_S	toast	0	60	4	-dfps -c [input].gsm > [output]
18	BE_bitcount_S	bitcnts	0	60	4	75000 > [output]
21	BE_sha2_S	sha	0	80	4	< [input].asc > [output].txt
22	BE_susan-S_co.	susan	0	80	1	-c [input].pgm [output].pgm
23	BE_susan-S_sm.	susan	0	80	8	-s [input].pgm [output].pgm
24	BE_susan-S_ed.	susan	0	80	8	-e [input].pgm [output].pgm
25	BE_cjpeg2_S	cjpeg	0	30	F	-dct int -progressive -opt [...]
26	BE_gsmToast2_S	toast	0	60	8	-fps -c [input].au > [output]
27	BE_gsmUToast2_S	toast	0	60	8	-dfps -c [input].gsm > [output]
28	BE_bitcount2_S	bitcnts	0	60	F	75000 > [output].txt

Code 5.4 – Tâches non critiques sélectionnées

En plus des informations déjà présentées précédemment sur le format d'un tel fichier de configuration, on remarquera ici la colonne de l'affinité des tâches, qui est exprimé en hexadécimal pour couvrir toutes les configurations possibles sur un processeur à 4 coeurs. Ainsi, par exemple F(hex) = 1111(binaire) donne l'affinité par défaut à l'ensemble des coeurs, tandis que 1 (=0001), 2 (=0010), 4 (=0100) et 8 (=1000) permettent un partitionnement des tâches à un cœur unique. Dans la configuration ci-présentée on a un système semi-partitionné, où la plupart des tâches sont assignées à un cœur unique tandis qu'une minorité sont libres de migrer entre tous les coeurs.

Cela nous permet de mettre en place l'étape ⑤ du protocole général, où l'on exécute à la fois la chaîne de tâches critiques et les tâches non critiques, accompagnées du mécanisme de surveillance sans la partie Contrôle. On désactive pour cela l'envoi du signal de changement de mode, mais tout le reste de l'algorithme du contrôle reste actif.

Cela nous permet d'une part de mesurer l'ordonnançabilité du système ainsi déployé. On peut aussi mesurer l'empreinte du Core Control Component en termes de temps d'exécution sur le CPU. Et enfin, en provoquant des déclenchements du mode dégradé de façon forcée (option directement en dur dans le code du Core Control Component) on peut mesurer le temps de changement de mode t_{SW} .

Le résultat que l'on obtient sur le profil d'exécution de la chaîne de tâche est compilé dans la Figure 5.10 avec la courbe rouge, superposée à la courbe bleue précédemment obtenue. On distingue clairement la conséquence d'une quasi surcharge du système avec l'ajout des tâches non critiques, qui provoquent des interférences réelles (en opposition avec les interférences forcées artificiellement via stress-ng) sur l'exécution des tâches critiques.

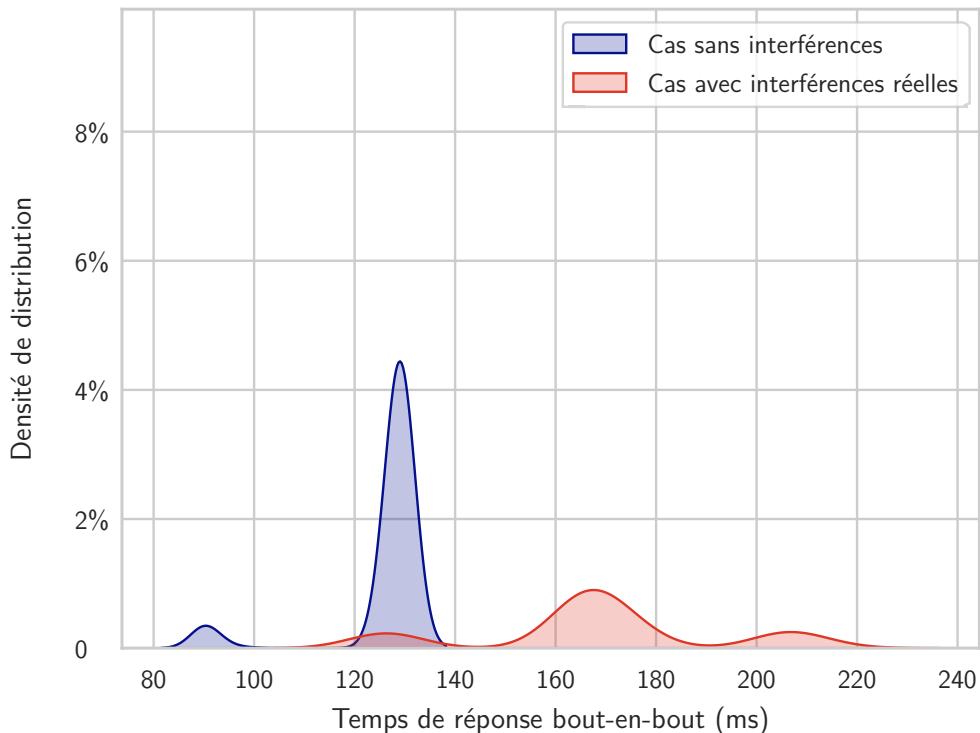


FIGURE 5.10 – Profil de la chaîne de tâche sans mécanisme de contrôle

On constate que le temps d'exécution moyen s'est déplacé autours des 170 ms, avec des pics plus rares dépassant les 200 ms dans environ 10% des mesures (l'échéance que nous avons fixé!). Le détail de ces valeurs est indiqué dans le Tableau 5.6, avec les écarts vis-à-vis du profil mesuré en isolation. Cet écart s'explique

pour l'essentiel à cause de l'utilisation intensive du cache L3 qui est partagé entre les coeurs, mais aussi pour partie parce qu'un certain nombre de tâches non critiques s'exécutent sur le même cœur que la chaîne de tâches critiques. Ainsi la combinaison de ce partage du processeur et de la congestion de l'utilisation du cache partagé provoque des temps de latence sur les tâches critiques, jusqu'à prendre plusieurs périodes de décalage sur la terminaison de la chaîne.

Cette différence est intéressante à noter du fait qu'elle quantifie l'écart entre les comportements pire-cas que l'on constate sur le système complet, avec le profil sur lequel on se base pour effectuer l'anticipation. Le principe même de se focaliser sur des temps d'exécution garantis en mode dégradé, qui sont plus faibles que les temps en fonctionnement nominal (c.-à-d. avec les tâches non critiques), permet au mécanisme de laisser plus de liberté aux tâches non critiques à l'exécution. Et ce parce que l'on se base sur des temps de réponse restant garanti en mode dégradé qui sont plus faible, en d'autre termes qui peuvent compenser un plus grand retard sur le temps de réponse bout-en-bout.

TABLE 5.6 – Profil de la chaîne de tâches critique avec interférences des tâches non critiques

	temps en isolation			temps nominaux		
	Min	Moyen	Max	Min	Moyen	Max
temps de réponse (ms)	65.41	125.03	138.26	125.51	168.41	214.98

Empreinte du mécanisme de contrôle – En ce qui concerne l'empreinte supplémentaire associée à l'ajout de notre mécanisme, elle semble négligeable. En effet, au fil des tests précédents, nous n'avons constaté aucune différence notable sur le taux d'utilisation du CPU avec et sans le Core Control Component. Aussi, lors des tests nous obtenons de façon très nette des mesures de temps d'exécution total du mécanisme qui se décomposent en deux parties :

- le temps d'exécution fixe qui inclut le temps de configuration de tout le système, qui sépare le moment où le processus est créé du moment où toutes les tâches sont lancées pour la durée du test ;
- le temps d'exécution variable pendant le fonctionnement de l'expérimentation en elle-même, et qui augmente normalement avec le temps de test. C'est celle-ci qui nous intéresse.

Nous mesurons une partie fixe très stable à deux paliers, soit entre 200ms et 240ms, soit entre 320 ms et 360 ms. La différence se fait très clairement selon s'il s'agit d'un test avec uniquement les tâches critiques ou bien aussi avec les tâches non critiques, ce qui permet de bien identifier le temps de configuration initial des tâches non critiques. On peut donc estimer en bonus le temps de configuration et création des processus des tâches non critiques qui se situe entre 80 ms et 160 ms.

Concernant la partie variable, les valeurs ont des variations très marginales quand on compare avec et sans l'inclusion des tâches non critiques. Le résultat est donc essentiellement influencé par le nombre de tâches critiques dans la chaîne. Pour

des tests de 100 secondes, on mesure ainsi autours de 340 ms de temps d'exécution du CCC à une période de 2 ms. Cela correspond à 0,34% de temps d'occupation. Si l'on reporte cette valeur sur la période d'exécution du mécanisme, on obtient aussi une durée d'exécution moyenne du CCC toutes les 2ms qui est de 68 μ s. Dans les faits, cette valeur varie s'il y a de nouvelles exécutions de tâches critiques à prendre en compte pour mettre à jour l'état de la chaîne, ou s'il n'y a qu'une nouvelle anticipation de changement de mode à calculer, ce qui est effectivement très rapide avec la structure de données et la méthode de calcul que nous avons considérée.

Temps de passage en mode dégradé – Enfin, avec les passages forcés en mode dégradé, on mesure un temps maximum mesuré de changement de mode t_{SW} . Il s'avère que les valeurs mesurées sont à la limite du négligeable. En effet, grâce à Xenomai et le pipeline de gestion d'interruptions ADEOS, la gestion des signaux de stop sont de l'ordre de quelques microsecondes. Étant donné les approximations de l'ordre de dixièmes de millisecondes sur les rWCRT précédemment, il nous semble par conséquent raisonnable de négliger cette valeur dans notre situation : $t_{SW} \approx 0$.

À ce stade, nous disposons donc d'un jeu de tâches complet incluant des tâches non-critiques qui provoquent des interférences et des tâches critiques qui constituent une chaîne de tâches. Les paramètres du mécanisme de Surveillance et Contrôle ont été configurés pour accomplir son rôle. Il demeure la phase de vérification avec l'activation du Contrôle et les mesures de performance.

5.3.3 Phase de Validation et Analyses

Cette dernière phase consiste à exécuter successivement la chaîne de tâches seule avec le mécanisme de contrôle, puis de faire de même tout en ajoutant les tâches non critiques. De cette façon, le premier résultat direct sera de comparer le profil des temps d'exécution de la chaîne de tâches critiques avec les précédentes mesures sans le mécanisme. Nous pourrons en plus de cela mener une analyse qualitative sur le mécanisme, qui se subdivise en 3 éléments : Fiabilité, Qualité et Performance.

- Tout d'abord la **fiabilité**, il s'agit de mesurer la propension du mécanisme à correctement prévenir les dépassements d'échéance bout-en-bout.
- Ensuite la **qualité**, où l'on regarde dans quelle mesure le mécanisme a tendance à déclencher des anticipations non nécessaires, autrement dit des faux-positifs, au détriment des tâches non critiques.
- Enfin, la **performance**, qui mesure le temps que l'on a pu conserver pour les tâches non critiques, ou bien vu dans l'autre sens, à quel point il a fallu mettre en pose les tâches non critiques pour atteindre l'objectif de fiabilité.

Dans un monde parfait, on souhaitera un mécanisme à 100% fiable, aux performances idéales qui ne stoppent jamais les tâches non critiques et donc à la qualité parfaite. Mais bien entendu comme nous ne sommes pas dans un monde parfait, on se doute que tout est histoire de compromis.

5.3.3.1 Chaîne de tâches avec mécanisme de Contrôle

On lance donc des tests avec uniquement la chaîne de tâches critiques comme charge utile, ainsi que le mécanisme de Surveillance et de Contrôle. Cela correspond à l'étape ⑥ On obtient dans ce cadre-là le même résultat que celui observé dans la Figure 5.8, mais avec la connaissance supplémentaire des déclenchements du mécanisme, alors même qu'il n'y a pas de risque de dépassement de l'échéance de temps de réponse.

Sur 1496 instanciations de la chaîne de tâche sur une durée de test de 100 secondes, nous avons mesuré 9 instants où le mécanisme obtient une estimation d'anticipation de risque positive, qui déclenche le passage en mode dégradé. Changement de mode qui n'a aucun effet, étant donné qu'il n'y a pas de tâches non critiques dans ce test, à dessein. Cela donne une mesure de **qualité** aux alentours de 0.6% d'anticipations non nécessaires dans notre configuration.

5.3.3.2 Système complet avec mécanisme de Contrôle

Pour finir, l'expérimentation finale consiste à exécuter la totalité du système. Cette étape ⑦ du protocole général implique donc tous les éléments précédemment définis, la chaîne de tâches critiques, les tâches non critiques, ainsi que le mécanisme de surveillance et de contrôle totalement opérationnel. Le résultat obtenu est représenté dans le graphique 5.11.

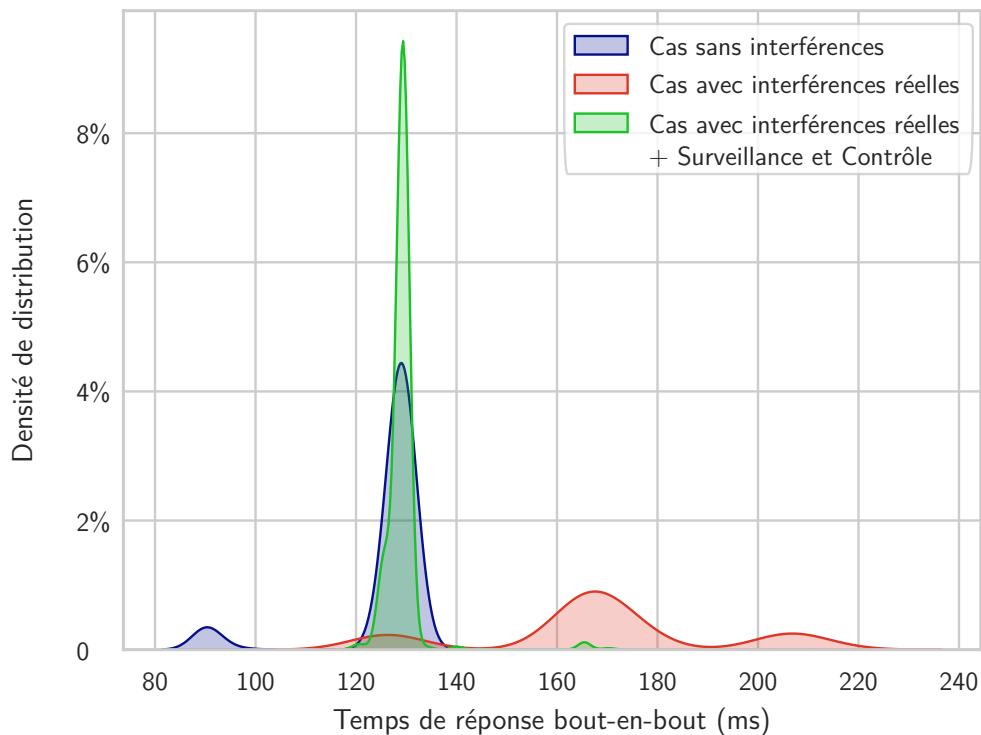


FIGURE 5.11 – Profil de la chaîne de tâche avec et sans mécanisme de contrôle

En terme d'**efficacité**, rappelons que l'échéance que nous avions fixé pour la chaîne de tâches est de 200 ms. Dans ce cadre-ci, il s'avère que notre système est effectivement à 100% efficace.

Il reste la question de la **performance**, sur l'exécution des tâches non critiques. Il s'avère que le mécanisme s'est activé 215 fois, pour 1376 occurrences complétées de la chaîne de tâches durant les 100 secondes de test. Cela représente 622 jobs de tâches non critiques qui ont dû être mis en pause en mode dégradé. Pour se représenter ce que cela représente, nous comparons aux résultats de l'étape ③, obtenus précédemment avec la chaîne de tâche et les tâches non critiques d'exécutées dans le mécanisme (courbe rouge du graphique 5.11). Si l'on additionne le temps cumulé d'exécution des tâches non critiques en l'absence de mécanisme, on obtient 346602,75 ms, soit environ 346,6 s⁴. Avec l'activation du mécanisme, en multipliant le nombre d'occurrences de tâches non critiques bloquées par leur temps d'exécution moyen, on obtient 65299,81 ms ≈ 65,29 s. En d'autres termes, le temps de blocage effectif de l'exécution des tâches non critiques est de l'ordre de 65 s dans ce cas particulier. En calculant le ratio des deux valeurs, on obtient alors la proportion du temps d'exécution des tâches non critiques qui ont été différées, qui est de 18,84%. Il s'agit d'une proportion non négligeable, mais qui semble être relativement cohérente avec la quantité assez importante d'échéances non respectées dans la situation initiale, qui était de l'ordre de 10%. Le compromis peut donc sembler envisageable dans ce cas précis. Il serait probablement d'autant plus intéressant dans le cadre d'un système qui est soumis à moins d'aléas et moins de variations de temps d'exécution, pour lequel le non-respect d'échéance en mode nominal restera hautement marginal et le mécanisme ne sera là qu'en ultime recours pour un évènement qui se voudra le plus rare possible.

Par ailleurs, on constate une petite part de valeurs dans la plage des 165 ms de temps de réponse bout-en-bout avec le mécanisme de contrôle actif. On peut donc se demander quelle serait l'influence d'une modification de la valeur d'échéance imposée dans le comportement du mécanisme. Il s'avère que nous avons effectué ce test, mais c'est là que le mécanisme montre ses limites. Dans une situation comme celle-ci où le système est dans un état à la limite de l'ordonnançable, avec un très fort impact des tâches non critiques sur l'exécution de la chaîne de tâches, il n'est pas possible de garantir une échéance d'exécution bout-en-bout aussi proche du profil d'exécution en isolation. D'une part cela provoque un déclenchement de l'anticipation quasi-systématique, du fait que les marges entre les temps de réponse avec interférences et les temps de réponse garantis en isolation sont faibles. Aussi, le contexte de notre plateforme matérielle, basée sur un système Linux grand-public, n'est pas favorable à la maîtrise complète de tout le code qui est exécuté par l'OS. Cela apporte donc potentiellement des aléas d'exécution, même en mode dégradé. Ainsi, avec un tel test où l'échéance a été réduite à 160 ms (ce qui est donc très exigeant comparé au profil de la chaîne), il y a eu malgré tout 30 exécutions qui se sont soldées par un dépassement de l'échéance, soit une proportion de 1,8% de fautes.

4. Note : sur un temps d'expérimentation de 100 secondes, c'est tout à fait cohérent avec la présence de 4 coeurs pour exécuter du code en parallèle, soit potentiellement 400 s de temps de calcul disponible.

Conclusion et Perspectives

Conclusion

Au fil de cette thèse, il nous a été donné de faire un état des lieux sur l'évolution des systèmes cyberphysiques, notamment dans le domaine de l'automobile. Ces évolutions se sont pour une grande part orientées vers le choix d'architectures matérielles multicœurs de plus en plus puissantes, mais aussi de plus en plus complexes. Avec la cohabitation de logiciels à différents niveaux de criticité sur ces plateformes, cela donne lieu à des enjeux émergents sur le respect des contraintes de sûreté de fonctionnement dans les systèmes temps-réel.

C'est donc en partant de ce constat que nous avons établi un état des lieux à la fois des enjeux et contraintes industrielles et des travaux afférents dans le domaine académique sur l'intégration de systèmes à criticité mixte. Les enjeux sont multiples et les solutions pour y répondre sont multicritères. Nous avons identifié deux grands enjeux lors de notre constat.

- Tout d'abord, la nécessité de garanties sur les temps d'exécution des tâches, et notamment les tâches critiques, de façon à ne pas dépasser des échéances liées aux contraintes temps-réel.
- Ensuite, le besoin croissant d'exploiter au maximum les ressources de calcul à disposition, de façon à diminuer considérablement le nombre de processeurs nécessaires au sein des systèmes embarqués.

La problématique principale pour répondre à ces deux enjeux réside dans leur antinomie. Les solutions dédiées à la garantie des contraintes temps-réel se font au détriment de l'optimisation de la puissance de calcul, tandis que l'agrégation de logiciels à criticité multiple provoque des problèmes d'interférences qui peuvent mener à des retards d'exécution indésirables. Ces problèmes sont intrinsèques à l'usage des calculateurs multicœurs, par la présence de ressources partagées tel que le cache, les bus de données ou encore les différentes entrées/sorties. Avec l'exécution concourante de tâches sur plusieurs coeurs, cet usage de ressources partagées entre les logiciels mène à des points de contentions qui augmentent les temps de réponses des tâches jusqu'au dépassement des échéances.

Pour répondre à cette problématique, nous avons en premier lieu proposé une approche qui se focalise sur une vision fonctionnelle du système. De fait, au sein d'un calculateur multicœur, les tâches qui s'exécutent sont en partie interconnectées pour réaliser des fonctions spécifiques. Cela se traduit à l'échelle du logiciel par ce que l'on définit comme des chaînes de tâches. Nous proposons par conséquent un modèle de mécanisme de Surveillance et de Contrôle spécialisé dans la garantie de contraintes temporelles lors de l'exécution d'une chaîne de tâches. Le modèle proposé se veut adapté au domaine industriel par sa généricité avec des hypothèses de modèle qui prennent en compte les contraintes industrielles, tel que la présence de logiciel en boite noire. Notre mécanisme repose sur la Surveillance de l'avancement d'exécution

d'une chaîne de tâche. Il anticipe l'instant où les interférences dues aux tâches non critiques présentent un risque irréversible de dépassement de l'échéance d'exécution bout-en-bout. Le cas échéant, l'exécution des tâches non critiques est Contrôlé via une mise en pause momentanée pour prévenir toute interférence supplémentaire et donc garantir le respect de l'échéance bout-en-bout.

Afin de mettre en place ce mécanisme, nous avons proposé tout un processus de développement qui s'inscrit dans une démarche expérimentale. Cette approche permet la caractérisation nécessaire d'un système à criticité mixte de façon à obtenir une meilleure connaissance de son comportement et notamment vis-à-vis des interférences d'exécution. Dans un second temps, il indique aussi les grandes étapes à intégrer dans le cadre de l'implémentation du mécanisme de Surveillance et Contrôle sur un système embarqué. Cela inclut la détermination des paramètres de fonctionnement propres au mécanisme, mais aussi la phase de vérification analytique avec une mesure des performances suivant trois critères pertinents (Fiabilité, Performance et Qualité) ainsi que des possibilités d'ajustements.

Enfin, nous avons mis en application cette démarche expérimentale sur une plate-forme dédiée. Dans cette optique, nous avons développé un framework expérimental complet. Ce dernier permet d'exécuter directement n'importe quel jeu de tâches sous forme de fonctions de librairies avec à notre mécanisme de Surveillance et de Contrôle. La plateforme matérielle et logicielle que nous proposons permet, par le biais de fichiers de configuration d'entrées, une forte adaptabilité de configuration et de modifications pour s'adapter selon 3 axes : le support matériel et logiciel bas-niveau (ici, nous avons choisis Xenomai sur un multicœur Intel) ; les hypothèses du modèle de tâches et de chaîne de tâches ; et les données d'entrée (le jeu de tâches, les paramètres du mécanisme, la charge de stress du calculateur). Grâce à tout cet ensemble, nous présentons ainsi une base de plateforme expérimentale minimale de façon à caractériser pour un contexte spécifique le profil de temps d'exécution d'un logiciel, la place des interférences inter-logiciel au sein de celui-ci, et l'apport de notre mécanisme de Surveillance et de Contrôle sur le respect des échéances bout-en-bout.

La mise en place de cette plateforme complète accompagnée des protocoles expérimentaux proposés a pu être éprouvé au travers de la constitution d'un cas de test fictif, par le biais des tâches du benchmark MiBench. Ce cas de test nous a permis d'appliquer la démarche dans sa totalité et d'illustrer les capacités de notre mécanisme dans un cas spécifique. Les résultats obtenus sont encourageants et permettent concrètement d'obtenir un système à criticité mixte avec un mécanisme de contrôle réactif qui offre aux tâches non critiques le temps nécessaires hors des instants d'exception où le respect d'échéance de la chaîne de tâches critiques doit continuer d'être garanti.

Dans un contexte comme celui du domaine automobile, où les multiples fonctions embarquées se retrouvent agrégés au sein de calculateurs toujours plus puissants, la problématique de co-existence de logiciels à différents niveaux de criticité devient inévitable. Quand des systèmes d'aide à la conduite (ADAS) tel que le Système de Changement de Ligne ou l'Avertissement d'Angle-Mort risquent de coexister avec

du logiciel non-critique comme le système radio ou la chauffe des sièges, le besoin de garanties d'exécution devient pressant. Dans l'ensemble, nous proposons une approche inédite pour le respect des échéances temporelles tout en conservant de bonnes performances sur l'exploitation des ressources de calcul. Notre solution de maîtrise des fautes temporelles dues aux interférences est réactive contrairement à ce qui se fait habituellement de façon statique dans les solutions industrielles. Nous tentons de palier aux risques d'interférences liées au partage de ressources de façon conservative, en limitant l'exécution des tâches non critiques uniquement en cas de nécessité absolue.

Ce processus expérimental propose globalement une nouvelle approche aux problématiques émergentes des systèmes à criticité mixte, qui prend en compte au mieux les contraintes propres aux systèmes industriels. Il s'agit d'un premier pas vers des approches orientées sur les chaînes de tâches, avec des outils pour la caractérisation des systèmes industriels et la mise en œuvre de ce type de mécanismes de sûreté de fonctionnement somme toute très lié aux enjeux du temps-réel.

Perspectives

L'approche qui a été développée dans cette thèse est particulièrement prometteuse. Elle ouvre des pistes de recherche axées sur l'exploitation de chaînes de tâches pour proposer un mécanisme réactif de garantie des contraintes temporelles. Cela constitue une base qui ne demande qu'à être étouffée et améliorée.

Les résultats expérimentaux que nous avons obtenus sont encourageants dans le sens où ils démontrent la capacité d'un tel mécanisme à offrir des garanties de respect d'échéance avec des compromis limités sur l'exécution des tâches non critiques. Cependant, ils sont spécifiques au système à criticité mixte auquel il est appliqué. Le diagnostic sur la présence d'interférences et leurs effets sur l'exécution des tâches critiques est hautement dépendant à la fois du support matériel et du comportement des tâches en elles-mêmes. L'implémentation d'un tel mécanisme est ainsi conditionné à la caractérisation du système dans lequel il s'inscrit.

En ce sens, la générnicité de la solution et son applicabilité à un grand nombre d'architectures tant matérielles que logicielles est une force. En effet, bien que refléchie en partant d'un contexte automobile, notre proposition a été élargie pour s'adapter à des cas industriels de systèmes à criticité mixte divers et variés.

La première piste d'amélioration réside dans l'extension de la proposition actuelle de façon à gérer plus d'une unique chaîne de tâches critiques. Il s'agit de l'extension la plus directe à prendre en compte, qui apporte un grand avantage sur les possibilités d'utilisations du mécanisme de surveillance et contrôle.

Considérant la proposition actuelle, l'enjeu à résoudre pour cela réside dans la capacité à obtenir des garanties de respect d'échéance dans un mode dégradé qui prend en compte plusieurs chaînes de tâches. Cela implique donc un plus grand nombre de tâches dans ce mode, et donc des interférences inter-chaînes. Cela ne pourrait se faire qu'à la condition de connaître avec précision ces interférences-là, et les prendre en compte dans les estimations de pire temps d'exécution restant de

chacune des chaînes de tâches critiques.

Une seconde piste de développement repose sur l'extension du modèle de tâches à criticité duale que nous avons employé. Bien qu'un bon nombre de systèmes reposent uniquement sur deux niveaux de criticité, l'extension à un plus grand nombre pourrait correspondre à une plus grande variété de cas d'applications. Ainsi, des domaines comme le ferroviaire, l'avionique ou l'automobile emploient 5 niveaux de criticité qui ont une incidence sur les contraintes de développement du logiciel. Proposer plusieurs niveaux de criticité aurait donc le double avantage d'être plus facilement combinable avec d'autres travaux de recherche qui travaillent sur plus de niveaux, et de potentiellement approfondir les possibilités de modes dégradés intermédiaires. En effet, les tâches non critiques peuvent alors être subdivisées dans des nuances de niveaux de criticité qui peuvent ne pas être traitées de la même façon par le mécanisme de contrôle.

Pour finir, dans l'idée de combiner notre mécanisme avec d'autres solutions de contrôle de l'exécution des tâches, il serait possible de proposer des niveaux intermédiaires de modes dégradés. Mais cette fois-ci il ne s'agirait pas de jouer sur quelles tâches sont mises en pause en mode dégradé, mais plutôt de proposer des solutions de contrôle moins drastiques, qui n'empêchent pas l'exécution des tâches non critiques, mais limitent par d'autres biais leurs capacités de nuisance par interférences. L'un de ces moyens repose sur l'exploitation du complément à notre méthode logicielle réactive : une solution matérielle statique. En effet, par la possibilité de mettre en place des mécanismes d'isolation spatiale des tâches en cours d'exécution, il serait possible de limiter les interférences sur les ressources partagées par la réservation d'une partie du cache aux tâches critiques. Une telle amélioration permettrait la mise en place de mécanismes préventifs progressifs contre les interférences entre les tâches. De façon cela limiterai au maximum la sous exploitation des ressources de calcul, tout en garantissant au mieux le respect des échéances temporelles sur les chaînes de tâches critiques.

ANNEXE 1 – Caractérisation des tâches MiBench

TABLE A1 – Caractérisation des tâches MiBench - 1200 runs par tâches

Tâche	Tâche en isolation			Tâche + interférences			Classification
	MIN (ms)	AVG (ms)	MAX (ms)	MIN (ms)	AVG (ms)	MAX (ms)	
fft-S	1,7E+00	1,7E+00	1,7E+00	1,7E+00	1,7E+00	1,9E+00	CLEAN
fft-inv-S	3,5E+00	3,5E+00	3,5E+00	3,5E+00	3,5E+00	3,7E+00	CLEAN
basicmath-S	3,8E+00	3,8E+00	4,0E+00	3,8E+00	3,9E+00	4,3E+00	CLEAN
fft-L	7,1E+00	7,1E+00	7,2E+00	6,7E+00	7,2E+00	7,7E+00	CLEAN
bitcount-S	7,3E+00	8,4E+00	9,8E+00	7,3E+00	8,5E+00	9,7E+00	CLEAN
basicmath-L	26,2E+00	26,3E+00	26,4E+00	26,3E+00	26,5E+00	27,8E+00	CLEAN
jpeg-D-S	1,9E+00	2,0E+00	14,0E+00	2,0E+00	2,2E+00	3,1E+00	NOISE
susan-corners-S	4,2E+00	4,3E+00	5,2E+00	4,3E+00	4,3E+00	4,6E+00	NOISE
susan-smooth-S	4,3E+00	4,3E+00	4,4E+00	4,3E+00	4,3E+00	5,1E+00	NOISE
susan-edges-S	4,3E+00	4,3E+00	4,4E+00	4,3E+00	4,3E+00	5,2E+00	NOISE
sha-S	4,7E+00	4,7E+00	4,7E+00	4,7E+00	4,8E+00	5,0E+00	NOISE
gsmToast-S	5,8E+00	5,8E+00	63,4E+00	6,0E+00	7,5E+00	15,6E+00	NOISE
gsmUToast-S	4,1E+00	4,2E+00	4,7E+00	4,7E+00	11,6E+00	40,4E+00	NOISE
fft-inv-L	14,6E+00	14,7E+00	14,8E+00	15,5E+00	15,7E+00	16,5E+00	NOISE
rijndael-E-S	12,9E+00	13,4E+00	63,6E+00	13,0E+00	18,9E+00	40,1E+00	NOISE
adpcm-C-S	27,4E+00	27,5E+00	38,8E+00	27,7E+00	36,9E+00	52,9E+00	NOISE
sha-L	48,7E+00	48,7E+00	48,9E+00	48,9E+00	49,3E+00	50,5E+00	NOISE
susan-corners-L	64,4E+00	64,3E+00	64,7E+00	57,4E+00	64,7E+00	65,9E+00	NOISE
susan-edges-L	64,4E+00	64,3E+00	64,7E+00	64,4E+00	64,7E+00	65,8E+00	NOISE
stringsearch-S	325,8E-03	342,3E-03	394,9E-03	366,5E-03	827,3E-03	10,2E+00	OVERHEAD
jpeg-C-S	10,2E+00	10,2E+00	13,6E+00	10,3E+00	10,7E+00	62,8E+00	OVERHEAD
rijndael-D-S	8,3E+00	12,9E+00	3,2E+03	8,4E+00	11,4E+00	25,5E+00	OVERHEAD
jpeg-D-L	6,3E+00	50,3E+00	302,4E+00	6,3E+00	13,8E+00	307,6E+00	OVERHEAD
jpeg-C-L	32,7E+00	33,6E+00	89,6E+00	29,8E+00	34,8E+00	106,0E+00	OVERHEAD
adpcm-D-S	13,3E+00	55,4E+00	5,4E+03	12,6E+00	41,E+00	6,9E+03	OVERHEAD
susan-smooth-L	64,4E+00	64,3E+00	64,8E+00	64,4E+00	64,7E+00	65,7E+00	OVERHEAD
qsort-S	17,6E-03	22,7E-03	24,1E-03	16,3E-03	21,3E-03	55,0E-03	REJECT
qsort-L	15,2E-03	22,6E-03	26,8E-03	17,5E-03	21,4E-03	31,5E-03	REJECT
patricia-L	16,2E-03	26,7E-03	100,8E-03	20,2E-03	25,9E-03	102,1E-03	REJECT
patricia-S	17,8E-03	26,6E-03	100,2E-03	18,7E-03	26,1E-03	103,8E-03	REJECT
blowfish-E-L	69,6E-03	69,7E-03	72,0E-03	71,0E-03	72,6E-03	75,3E-03	REJECT
blowfish-D-S	69,6E-03	69,7E-03	71,8E-03	70,8E-03	72,7E-03	75,2E-03	REJECT
blowfish-D-L	69,6E-03	69,8E-03	70,1E-03	70,6E-03	72,7E-03	76,4E-03	REJECT
blowfish-E-S	69,4E-03	69,5E-03	77,9E-03	71,2E-03	72,9E-03	75,7E-03	REJECT
rijndael-D-L	90,3E+00	109,2E+00	6,9E+03	80,6E+00	149,2E+00	11,9E+03	REJECT
adpcm-D-L	228,9E+00	355,6E+00	5,6E+03	228,8E+00	395,9E+00	7,2E+03	REJECT
gsmToast-L	281,9E+00	287,3E+00	397,2E+00	449,6E+00	518,4E+00	577,5E+00	REJECT
adpcm-C-L	411,7E+00	559,7E+00	11,3E+03	662,8E+00	801,7E+00	2,3E+03	REJECT
gsmUToast-L	199,9E+00	204,3E+00	455,2E+00	687,6E+00	907,3E+00	1,1E+03	REJECT
rijndael-E-L	127,7E+00	152,3E+00	5,8E+03	1,2E+03	1,5E+03	1,7E+03	REJECT
Médianes	7,83 ms			9,64 ms			

Bibliographie

- [Allende 2019] Allende, I., Mc Guire, N., Perez, J., Monsalve, L. G., Uriarte, N. et Obermaisser, R. *Towards Linux for the Development of Mixed-Criticality Embedded Systems Based on Multi-Core Devices*. Dans 2019 15th European Dependable Computing Conference (EDCC), pages 47–54, Naples, Italy, 2019. IEEE. (Cité en pages 75 et 82.)
- [Anderson 2009] Anderson, J. H., Baruah, S. K. et Brandenburg, B. B. *Multicore Operating-System Support for Mixed Criticality*. Dans Workshop on Mixed Criticality : Roadmap to Evolving UAV Certification, volume 4. Citeseer, 2009. (Cité en page 33.)
- [Augier 2006] Augier, C., Armand, F. et Muller, G. *Real-Time Scheduling in a Virtualized Environment*. Rapport technique, Department of Computer Science YCS, York, UK, 2006. (Cité en page 31.)
- [AUTOSAR 2016] AUTOSAR. *Timing Analysis*. Standard Release 4.3.0, 2016. (Cité en page 32.)
- [Avizienis 2004] Avizienis, A., Laprie, J.-C., Randell, B. et Landwehr, C. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pages 11–33, 2004. [Online]. Available: <https://dx.doi.org/10.1109/TDSC.2004.2>. (Cité en page 14.)
- [Baufreton 2010] Baufreton, P., Blanquart, J., Boulanger, J., Delsenay, H., Derrien, J., Gassino, J., Ladier, G., Ledinot, E., Leeman, M., Quéré, P. et Ricque, B. *Multi-Domain Comparison of Safety Standards*. Embedded Real Time Software & Systems (ERTS2), 2010. (Cité en page 20.)
- [Bechennec 2006] Bechennec, J.-L., Briday, M., Faucou, S. et Trinquet, Y. *Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification*. Dans Conference on Emerging Technologies and Factory Automation, pages 62–69. IEEE, 2006. (Cité en page 31.)
- [Behera 2012] Behera, L. et Mohapatra, D. P. *Schedulability Analysis of Task Scheduling in Multiprocessor Real-Time Systems Using EDF Algorithm*. Dans International Conference on Computer Communication and Informatics (ICCCI). IEEE, 2012. (Cité en page 35.)
- [Blanchet 2016] Blanchet, M. *Industrie 4.0 : nouvelle donne industrielle, nouveau modèle économique*. Géoéconomie, vol. 82, no. 5, page 37, 2016. [Online]. Available: <https://dx.doi.org/10.3917/geoec.082.0037>. (Cité en page 6.)
- [Bletsas 2018] Bletsas, K., Awan, M. A., Souto, P. F., Akesson, B., Burns, A. et Tovar, E. *Decoupling Criticality and Importance in Mixed-Criticality Scheduling*. Workshop on Mixed Criticality, page 7, 2018. (Cité en page 42.)
- [Blin 2016a] Blin, A., Courtaud, C., Sopena, J., Lawall, J. et Muller, G. *Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System*. Dans 28th Euromicro Conference on Real-Time Systems (ECRTS), pages 109–119. IEEE, 2016. (Cité en pages 34 et 65.)

- [Blin 2016b] Blin, A., Courtaud, C., Sopena, J., Lawall, J. et Muller, G. *Understanding the Memory Consumption of the MiBench Embedded Benchmark*. Dans International Conference on Networked Systems, pages 71–86, Marakech, Morocco, 2016. (Cité en page 88.)
- [Blin 2017] Blin, A. *Vers une utilisation efficace des processeurs multi-coeurs dans des systèmes embarqués à criticités multiples*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Paris, 2017. (Cité en page 30.)
- [Boniol 2019] Boniol, F., Pagetti, C. et Sensfelder, N. *Identification of Multi-Core Interference*. Dans 2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE), pages 98–106, 2019. (Cité en page 80.)
- [Bouchier 2013] Bouchier, P. *Embedded ROS*. IEEE Robotics Automation Magazine, vol. 20, no. 2, pages 17–19, 2013. [Online]. Available: <https://dx.doi.org/10.1109/MRA.2013.2255491>. (Cité en page 82.)
- [Brown 2010] Brown, D. J. H. et Martin, B. *How Fast Is Fast Enough ? Choosing between Xenomai and Linux for Real-Time Applications*. Dans Twelfth Real-Time Linux Workshop, Nairobi, 2010. (Cité en page 83.)
- [Burns 2022] Burns, A. et Davis, R. I. *Mixed Criticality Systems - A Review*. Rapport technique, Department of Computer Science, University of York, York, UK, 2022. (Cité en page 29.)
- [Charara 2006] Charara, H., Scharbarg, J.-L., Ermont, J. et Fraboul, C. *Methods for Bounding End-to-End Delays on an AFDX Network*. Dans 18th Euromicro Conference on Real-Time Systems (ECRTS'06), pages 10 pp.–202, 2006. (Cité en page 32.)
- [Chen 2014] Chen, Y., Li, Q., Li, Z. et Xiong, H. *Efficient Schedulability Analysis for Mixed-Criticality Systems under Deadline-Based Scheduling*. Chinese Journal of Aeronautics, vol. 27, no. 4, pages 856–866, 2014. [Online]. Available: <https://dx.doi.org/10.1016/j.cja.2014.05.003>. (Cité en page 68.)
- [Chisholm 2017] Chisholm, M., Kim, N., Tang, S., Otterness, N., Anderson, J. H., Smith, F. D. et Porter, D. E. *Supporting Mode Changes While Providing Hardware Isolation in Mixed-Criticality Multicore Systems*. Dans 25th International Conference on Real-Time Networks and Systems, pages 58–67, Grenoble France, 2017. ACM. (Cité en page 36.)
- [Durrieu 2014] Durrieu, G., Faugere, M., Girbal, S., Pérez, D. G., Pagetti, C. et Puffitsch, W. *Predictable Flight Management System Implementation on a Multicore Processor*. Dans Embedded Real Time Software (ERTS'14), 2014. (Cité en page 10.)
- [Fleming 2013] Fleming, T. et Burns, A. *Extending Mixed Criticality Scheduling*. Dans RTSS, 2013. (Cité en page 42.)
- [Friese 2018] Friese, M. J., Ehlers, T. et Nowotka, D. *Estimating Latencies of Task Sequences in Multi-Core Automotive ECUs*. Dans 2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES), pages 1–10, Graz, Austria, 2018. IEEE. (Cité en page 48.)

- [Gerum 2004] Gerum, P. *Xenomai - Implementing a RTOS Emulation Framework on GNU/Linux*. Rapport technique, Xenomai, 2004. (Cité en page 82.)
- [Gerum 2005] Gerum, P. *Life with Adeos*. Xenomai, Munich, Germany, Tech. Rep, 2005. [Online] (Cité en page 83.)
<https://www.xenomai.org/documentation/branches/v2.2.x/pdf/life-with-adeos.pdf>.
- [Gerum 2015] Gerum, P. *Xenomai 3 : hybride et caméléon*. Open Silicium, no. OS-016, 2015. [Online] (Cité en page 85.)
<https://connect.ed-diamond.com/Open-Silicium/os-016/xenomai-3-hybride-et-cameleon>.
- [Giannopoulou 2013] Giannopoulou, G., Stoimenov, N., Huang, P. et Thiele, L. *Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems*. Dans ACM International Conference on Embedded Software, 2013. (Cité en page 32.)
- [Girbal 2018] Girbal, S., Le Rhun, J. et Saoud, H. *METRICS : A Measurement Environment for Multi-Core Time Critical Systems*. Dans ERTS 2018, page 10, 2018. (Cité en page 82.)
- [Gobillot 2018] Gobillot, N. et Lucet, E. *ESPRIT : Overview of the Vehicles Road-Train Real-Time Architecture*. Dans ERTS 2018, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, 2018. (Cité en page 82.)
- [Graydon 2013] Graydon, P. et Bate, I. *Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling*. Dans WMC, 2013. (Cité en page 41.)
- [Guthaus 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T. et Brown, R. B. *MiBench : A Free, Commercially Representative Embedded Benchmark Suite*. Dans 4th International Workshop on Workload Characterization, Austin, TX, USA, 2001. IEEE. (Cité en page 86.)
- [Han 2018] Han, J.-J., Tao, X., Zhu, D., Aydin, H., Shao, Z. et Yang, L. T. *Multicore Mixed-Criticality Systems : Partitioned Scheduling and Utilization Bound*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 1, pages 21–34, 2018. [Online]. Available: <https://dx.doi.org/10.1109/TCAD.2017.2697955>. (Cité en page 68.)
- [Herman 2012] Herman, J. L., Kenna, C. J., Mollison, M. S., Anderson, J. H. et Johnson, D. M. *RTOS Support for Multicore Mixed-Criticality Systems*. Dans 18th Real Time and Embedded Technology and Applications Symposium, pages 197–208. IEEE, 2012. (Cité en page 33.)
- [IEC 61508 2010] IEC 61508. *Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*. Rapport technique IEC 61508, The International Electrotechnical Commission, 2010. (Cité en page 19.)
- [Intel Corporation 2009] Intel Corporation. *Hard Real Time Linux Using Xenomai on Intel Multi-Core Processors*. White Paper, Intel Corporation, 2009. (Cité en page 83.)

- [Ishkov 2015] Ishkov, N. *A Complete Guide to Linux Process Scheduling*. M.Sc. Thesis, University of Tampere, 2015. (Cité en page 85.)
- [ISO 26262 Sec. 7 2018] ISO 26262 Sec. 7. *Road vehicles — Functional safety — Part 7 : Production, operation, service and decommissioning*. 2018. [Online] (Cité en page 20.)
<https://www.iso.org/cms/render/live/fr/sites/isoorg/contents/data/standard/06/83/68389.html>.
- [ISO TC22/SC3/WG16 2011] ISO TC22/SC3/WG16. *ISO 26262 : Road Vehicles — Functional Safety*. 2011. [Online] (Cité en page 19.)
<https://www.iso.org/fr/standard/68391.html>.
- [Juliussen 2022] Juliussen, E. *Navigating the Complexities of Software-Defined Vehicles*. 2022. [Online] (Cité en page 6.)
<https://www.embedded.com/navigating-the-complexities-of-software-defined-vehicles/>.
- [Kaiser 2007] Kaiser, R. et Wagner, S. *Evolution of the PikeOS Microkernel*. Dans First International Workshop on Microkernels for Embedded Systems, volume 50, 2007. (Cité en page 31.)
- [Kästner 2019] Kästner, D., Pister, M., Wegener, S. et Ferdinand, C. *Time Weaver : A Tool for Hybrid Worst-Case Execution Time Analysis*. Dans Altmeyer, S., éditeur, 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019), volume 72 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cité en page 31.)
- [Kim 2019] Kim, Y., More, A., Shriver, E. et Rosing, T. *Application Performance Prediction and Optimization Under Cache Allocation Technology*. Dans 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1285–1288, mars 2019. (Cité en page 34.)
- [King 2019] King, C. I. *Stress-Ng - A Stress-Testing Swiss Army Knife*. 2019. [Online] (Cité en pages 65, 82 et 88.)
<https://kernel.ubuntu.com/~cking/stress-ng/>.
- [Kotaba 2013] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, S. M. et Theiling, H. *Multicore in Real-Time Systems—Temporal Isolation Challenges Due to Shared Resources*. Dans 16th Design, Automation & Test in Europe Conference and Exhibition, 2013. (Cité en page 16.)
- [Kritikakou 2014] Kritikakou, A., Pagetti, C., Baldellon, O., Roy, M. et Rochange, C. *Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems*. Dans 26th Euromicro Conference on Real-Time Systems (ECRTS14), pages 119–128. IEEE, 2014. (Cité en page 53.)
- [Kritikakou 2016] Kritikakou, A., Marty, T., Pagetti, C., Rochange, C., Lauer, M. et Roy, M. *Multiplexing Adaptive with Classic AUTOSAR ? Adaptive Software Control to Increase Resource Utilization in Mixed-Critical Systems*. Dans Workshop CARS 2016-Critical Automotive Applications : Robustness & Safety, 2016. (Cité en page 36.)

- [Kritikakou 2017] Kritikakou, A., Marty, T. et Roy, M. *DYNASCORE : DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems.* ACM Transactions on Design Automation of Electronic Systems, vol. 23, no. 2, 2017. [Online]. Available: <https://dx.doi.org/10.1145/3110222>. (Cité en page 36.)
- [Laprie 1996] Laprie, J., Arlat, J., Blanquart, J., Costes, A., Abdeddaim, Y., Deswartre, Y., Fabre, J., Guillermain, H., Kaaniche, M., Kanoun, K., Mazet, C., Power, D., Rabejac, C. et Thevenod, P. Guide de la sûreté de fonctionnement. Cépaduès-Editions, France, 1996. (Cité en page 12.)
- [Lelli 2011] Lelli, J., Lipari, G., Faggioli, D. et Cucinotta, T. *An Efficient and Scalable Implementation of Global EDF in Linux.* 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11), 2011. (Cité en pages 35 et 86.)
- [Li 2014] Li, J., Chen, J. J., Agrawal, K., Lu, C., Gill, C. et Saifullah, A. *Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks.* Dans 2014 26th Euromicro Conference on Real-Time Systems, pages 85–96, Madrid, Spain, 2014. IEEE. (Cité en page 35.)
- [Li 2016] Li, S. et Mishra, S. *Optimizing Power Consumption in Multicore Smartphones.* Journal of Parallel and Distributed Computing, vol. 95, pages 124–137, septembre 2016. [Online]. Available: <https://dx.doi.org/10.1016/j.jpdc.2016.02.004>. (Cité en page 28.)
- [Lin 2006] Lin, J. D. et M. K. Cheng, A. *Maximizing Guaranteed QoS in (m, k)-Firm Real-time Systems.* Dans 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06), pages 402–410, San Jose, USA, 2006. IEEE Technical Committee on Real-Time Systems. (Cité en page 75.)
- [Lin 2015] Lin, J. D., Cheng, A. M. K., Steel, D., Wu, M. Y.-C. et Sun, N. *Scheduling Mixed-Criticality Real-Time Tasks in a Fault-Tolerant System :* International Journal of Embedded and Real-Time Communication Systems, vol. 6, no. 2, pages 65–86, 2015. [Online]. Available: <https://dx.doi.org/10.4018/IJERTCS.2015040104>. (Cité en page 37.)
- [Litayem 2011] Litayem, N., Ben, A. et Ben, S. *Building XenoBuntu Linux Distribution for Teaching and Prototyping Real-Time Operating Systems.* International Journal of Advanced Computer Science and Applications, vol. 2, no. 2, 2011. [Online]. Available: <https://dx.doi.org/10.14569/IJACSA.2011.020201>. (Cité en page 82.)
- [Lozi 2016] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V. et Fedorova, A. *The Linux Scheduler : A Decade of Wasted Cores.* Dans Proceedings of the Eleventh European Conference on Computer Systems, pages 1–16, London United Kingdom, 2016. ACM. (Cité en pages 28 et 82.)
- [Mancuso 2013] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M. et Pellizzoni, R. *Real-Time Cache Management Framework for Multi-Core Architectures.* Dans 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 45–54, 2013. (Cité en page 32.)

- [Melani 2016] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A. et Buttazzo, G. *Schedulability Analysis of Conditional Parallel Task Graphs in Multicore Systems*. IEEE Transactions on Computers, pages 1–1, 2016.
 [Online]. Available: <https://dx.doi.org/10.1109/TC.2016.2584064>. (Cité en page 68.)
- [Owens 2008] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. et Phillips, J. C. *GPU Computing*. Proceedings of the IEEE, vol. 96, no. 5, pages 879–899, 2008. (Cité en page 11.)
- [Pabla 2009] Pabla, C. S. *Completely Fair Scheduler*. Linux J., vol. 2009, no. 184, 2009. [Online] (Cité en pages 28 et 85.)
<http://dl.acm.org/citation.cfm?id=1594371.1594375>.
- [Pathania 2016] Pathania, A., Venkataramani, V., Shafique, M., Mitra, T. et Henkel, J. *Distributed Fair Scheduling for Many-Cores*. Dans DATE, pages 379–384, mars 2016. (Cité en page 28.)
- [Pricopi 2014] Pricopi, M. et Mitra, T. *Task Scheduling on Adaptive Multi-Core*. IEEE Transactions on Computers, vol. 63, no. 10, pages 2590–2603, 2014.
 [Online]. Available: <https://dx.doi.org/10.1109/TC.2013.115>. (Cité en page 28.)
- [Prisaznuk 2008] Prisaznuk, P. J. *ARINC 653 Role in Integrated Modular Avionics (IMA)*. Dans IEEE/AIAA 27th Digital Avionics Systems Conference, 2008. (Cité en page 32.)
- [Rodriguez 2013] Rodriguez, P., George, L., Abdeddaim, Y. et Goossens, J. *Multicriteria Evaluation of Partitioned Edf-vd for Mixed-Criticality Systems upon Identical Processors*. Dans Workshop on Mixed Criticality Systems, 2013. (Cité en page 35.)
- [Rupp 2020] Rupp, K. *42 Years of Microprocessor Trend Data / Karl Rupp*. 2020. [Online] (Cité en page 8.)
<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>.
- [Schmidt 2010] Schmidt, A., Dey, A. K., Kun, A. L. et Spiessl, W. *Automotive User Interfaces : Human Computer Interaction in the Car*. Dans Extended Abstracts on Human Factors in Computing Systems, pages 3177–3180. ACM, 2010. (Cité en page 6.)
- [Schoeberl 2011] Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F. et Probst, C. W. *Towards a Time-predictable Dual-Issue Microprocessor : The Patmos Approach*. Dans Bringing Theory to Practice : Predictability and Performance in Embedded Systems, volume 18, pages 11–21, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cité en page 30.)
- [Serra 2020] Serra, G., Ara, G., Fara, P. et Cucinotta, T. *An Architecture for Declarative Real-Time Scheduling on Linux*. Dans 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), pages 20–28, 2020. (Cité en pages 75 et 82.)

- [Sivakumar 2020] Sivakumar, P., Sandhya Devi, R. S., Neeraja Lakshmi, A., VinothKumar, B. et Vinod, B. *Automotive Grade Linux Software Architecture for Automotive Infotainment System*. Dans 2020 International Conference on Inventive Computation Technologies (ICICT), pages 391–395, Coimbatore, India, 2020. IEEE. (Cité en pages 75 et 82.)
- [Smotherman 2005] Smotherman, M. *History of Multithreading*. Retrieved on, pages 12–19, 2005. (Cité en page 8.)
- [Solet 2018] Solet, D., Pillement, S., Bechennec, J.-L., Briday, M. et Faucou, S. *HW-based Architecture for Runtime Verification of Embedded Software on SoPC Systems*. Dans 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pages 249–256, Edinburgh, 2018. IEEE. (Cité en page 30.)
- [Suhendra 2008] Suhendra, V. et Mitra, T. *Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores*. Dans Proceedings of the 45th Annual Conference on Design Automation - DAC '08, page 300, Anaheim, California, 2008. ACM Press. (Cité en pages 32 et 34.)
- [Sundar 2019] Sundar, V. K. et Easwaran, A. *A Practical Degradation Model for Mixed-Criticality Systems*. Dans 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), pages 171–180, 2019. (Cité en page 42.)
- [Sysgo AG 2019] Sysgo AG. *ARINC 653 RTOS for Multicore Certification*. White Paper, SysGo, 2019. (Cité en page 31.)
- [Tamas-Selicean 2011] Tamas-Selicean, D. et Pop, P. *Task Mapping and Partition Allocation for Mixed-Criticality Real-Time Systems*. Dans 17th Pacific Rim International Symposium on Dependable Computing, pages 282–283. IEEE, 2011. (Cité en page 32.)
- [Thompson 2006] Thompson, S. E. et Parthasarathy, S. *Moore's Law : The Future of Si Microelectronics*. Materials Today, vol. 9, no. 6, pages 20–25, 2006. [Online]. Available: [https://dx.doi.org/10.1016/S1369-7021\(06\)71539-5](https://dx.doi.org/10.1016/S1369-7021(06)71539-5). (Cité en page 8.)
- [Trapp 2007] Trapp, M., Adler, R., Förster, M. et Junger, J. *Runtime Adaptation in Safety-Critical Automotive Systems*. Dans 25th Conference on IASTED International Multi-Conference : Software Engineering, SE'07, pages 308–315, Innsbruck, Austria, 2007. ACTA Press. (Cité en page 36.)
- [Vestal 2007] Vestal, S. *Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance*. Dans 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pages 239–243, Tucson, AZ, USA, 2007. IEEE. (Cité en pages 31 et 41.)
- [Walsh 2004] Walsh, W., Tesauro, G., Kephart, J. et Das, R. *Utility Functions in Autonomic Systems*. Dans International Conference on Autonomic Computing, 2004. Proceedings., pages 70–77, New York, NY, USA, 2004. IEEE. (Cité en page 28.)

- [Wilkes 1965] Wilkes, M. V. *Slave Memories and Dynamic Storage Allocation*. IEEE Transactions on Electronic Computers, no. 2, pages 270–271, 1965. (Cité en page 10.)
- [Wong 2008] Wong, C. S., Tan, I., Kumari, R. D. et Wey, F. *Towards Achieving Fairness in the Linux Scheduler*. SIGOPS Oper. Syst. Rev., vol. 42, no. 5, pages 34–43, 2008. [Online]. Available: <https://dx.doi.org/10.1145/1400097.1400102>. (Cité en page 85.)
- [Xu 2019] Xu, H. et Burns, A. *A Semi-Partitioned Model for Mixed Criticality Systems*. Journal of Systems and Software, vol. 150, pages 51–63, 2019. [Online]. Available: <https://dx.doi.org/10.1016/j.jss.2019.01.015>. (Cité en page 35.)
- [Yun 2012] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M. et Sha, L. *Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality*. Dans 2012 24th Euromicro Conference on Real-Time Systems, pages 299–308, Pisa, Italy, 2012. IEEE. (Cité en page 35.)

Publications

Les travaux présentés dans ce manuscrit ont été effectués à Toulouse au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS) au sein de l'équipe Tolérance aux Fautes et Sûreté de Fonctionnement. Ils ont été réalisés dans le cadre d'une collaboration CIFRE avec le Technocentre Renault, Guyancourt et plus spécifiquement l'équipe SW-T. À l'heure à laquelle ces lignes sont écrites, cela a donné lieu aux publications suivantes :

- D. Loche, M. Lauer, M. Roy, et J.-C. Fabre, « *Mixed Critical Automotive Embedded Applications on Multicores : A Safe Scheduling Approach for Dependability* ». Dans 5th International Workshop on Critical Automotive Applications : Robustness & Safety (CARS), 2019, Naples, Italy
- D. Loche, M. Lauer, M. Roy, et J.-C. Fabre, « *Safe Scheduling on Multicores : an approach leveraging multi-criticality and end-to-end deadlines* ». Dans 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020), 2020, Toulouse, France.
- D. Loche, A. Generes, M. Lauer, et J.-C. Fabre, « *Run-time Monitoring and Control for Temporal Fault Prevention in Mixed-criticality Systems* ». Dans European Dependable Computing Conference (EDCC 2021), Intel ; Fraunhofer IKS ; LAAS, 2021, Munich (virtual), Germany. pp.53-60. <doi : 10.1109/EDCC53658.2021.00015>

Résumé :

L'émergence de calculateurs multicœurs plus puissants, mais aussi plus complexes, présente à la fois une opportunité et de nouveaux enjeux pour l'intégration de systèmes à criticité mixte. Ces systèmes, qui embarquent des fonctionnalités de différents niveaux d'importance, requièrent le respect d'exigences temporelles. En effet, plus un logiciel est critique, plus les conséquences en cas de dépassement d'échéance d'exécution peuvent être catastrophiques. Par conséquent, des mécanismes de sûreté de fonctionnement doivent être mis en place pour prévenir ce risque.

L'usage de calculateurs multicœurs exacerbe cette problématique avec l'exécution concourante de tâches qui peut provoquer des interférences inter-tâches. Cela est dû au partage de ressources associées à la plateforme matérielle partagée. La mémoire, les périphériques, les bus d'accès ainsi que les espaces mémoire partagés constituent ainsi des points de congestion potentielle. Cela peut alors mener à des retards dans l'exécution de tâches critiques uniquement du fait de l'exécution parallèle de tâches à plus faible niveau de criticité. En conséquence, les risques de non-respect des exigences temps-réel deviennent d'autant plus importants sur ces processeurs. De plus, la complexité croissante des architectures matérielle rend aussi plus difficile de complètement maîtriser et prévenir en amont tout risque d'interférences.

Les solutions existantes à un tel problème se font au détriment d'autres éléments qui deviennent tout autant nécessaires dans l'industrie, dont notamment la maximisation d'usage des ressources de calcul. Ainsi, notre proposition cherche à répondre à la fois au besoin de garanties temps-réel sur les tâches critiques tout en permettant une bonne utilisation des ressources offertes par les calculateurs multicœurs. Dans ce cadre, ces travaux se focalisent sur l'étude des tâches critiques sous forme de chaînes de tâches fonctionnelles. De fait chaque fonctionnalité exécutée correspond à une chaîne de différents blocs logiciels exécutés séquentiellement. Avec cette approche, on propose un mécanisme de Surveillance et de Contrôle qui offre des garanties minimales sur une chaîne de tâche critique, par anticipation de risques d'interférences qui pourraient mener à une défaillance. L'anticipation ayant pour moyen de neutralisation des interférences la mise en pause temporaire des tâches non critiques. L'objectif est d'obtenir une garantie d'exécution bout-en-bout de la chaîne de tâche sans dépassement d'échéance, avec une limitation des tâches non critiques uniquement quand nécessaire, de façon à ce que le reste du temps, on puisse exploiter au maximum la puissance de calcul et les ressources disponibles. Avec un tel mécanisme, nous proposons un protocole d'implémentation et de calibration, ainsi qu'une première analyse à partir d'une plateforme expérimentale dédiée

Mots clés : multicœur, temps-réel, criticité mixte, sûreté de fonctionnement

Abstract :

The emergence of more powerful, but also more complex, multicore computers presents both an opportunity and new challenges for the integration of mixed criticality systems. These systems, which embed functionalities of different levels of importance, require the respect of temporal requirements. Indeed, the more critical a software is, the more catastrophic the consequences in case of a missed execution deadline. Consequently, mechanisms for operational safety must be put in place to prevent this risk.

The use of multicore computers exacerbates this problem with the concurrent execution of tasks that can cause inter-task interference. This is due to the sharing of resources associated with the shared hardware platform. Memory, peripherals, access buses and shared memory spaces are potential congestion points. This can lead to delays in the execution of critical tasks only because of the parallel execution of lower criticality tasks. As a consequence, the risks of non-compliance with real-time requirements become even more important on these processors. Moreover, the increasing complexity of hardware architectures also makes it more difficult to completely control and prevent any risk of interference upstream.

The existing solutions to such a problem are done at the expense of other elements that are becoming equally necessary in the industry, including the maximization of the use of computing resources. Thus, our proposal seeks to address both the need for real-time guarantees on critical tasks while allowing a good use of the resources offered by multicore computers. This thesis work focuses on the study of critical tasks modeled as functional task chains to guarantee their response times in a mixed-criticality hosting multicore. Each functionality of the system corresponds to a chain of different software components executed sequentially. With this more functional approach to software execution, we want to obtain a Monitoring and Control mechanism that offers guarantees on a critical task chain, by anticipating risks of interference that could lead to a failure. The anticipation consisting of pausing temporarily the noncritical tasks to control and mitigate interference. The objective is to obtain a guarantee of end-to-end response time of the task chain without missing deadlines, with a limitation of interference only when necessary, so that the rest of the time, all the tasks of the system can exploit the computing power and the available resources to the maximum. We propose along with such mechanism an implementation protocol and a first analysis with an experimental platform as a proof of concept to analyze the results obtained with such solution.

Keywords : multicore, real-time, mixed-criticality, safety
